Erica Sadun

Das große iOS Entwicklerbuch

Rezepte für die App-Programmierung mit dem iOS SDK

ADDISON-WESLEY

ALWAYS LEARNING

PEARSON

Das große iOS-Entwicklerbuch

Ich widme dieses Buch in Liebe meinem Ehemann Alberto,
der in all den Jahren so viele technische Spielereien und SDKs ertragen musste
und dabei unterm Strich stets freundlich und geduldig blieb.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

10 9 8 7 6 5 4 3 2 1

13 12

ISBN: 978-3-8273-3063-5 (print); 978-3-86324-585-6 (PDF); 978-386324-173-5 (ePUB)

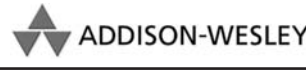
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Lektorat: Boris Karnikowski, bkarnikowski@pearson.de
Fachlektorat: Philipp Homann, wwwPage.de
Korrektorat: Friederike Daenecke, Zülpich
Covergestaltung: Marco Lindenbeck, mlindenbeck@webwo.de
Herstellung: Philipp Burkart, pburkart@pearson.de
Übersetzung und Satz: G&U Language & Publishing Services GmbH (www.GundU.com)
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

Erica Sadun

Das große iOS-Entwicklerbuch

Rezepte für die App-Programmierung mit dem iOS SDK



An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Ansichten und Animationen zusammenstellen

Die Klasse `UIView` und ihre Unterklassen dienen dazu, den Bildschirm von iOS-Geräten zu füllen. In diesem Kapitel werden Sie von Grund auf in den Umgang mit Ansichten eingeführt. Sie lernen, wie Sie Ansichtshierarchien erstellen, untersuchen und zerlegen und wie Ansichten zusammenwirken. Außerdem erfahren Sie, welche Rolle die Geometrie beim Erstellen und Platzieren der Ansichten in der Schnittstelle spielt und wie Sie Ansichten animieren, sodass sie sich auf dem Bildschirm bewegen und umwandeln. Dieses Kapitel behandelt von den Grundlagen an alles, was Sie über die Arbeit mit Ansichten wissen müssen.

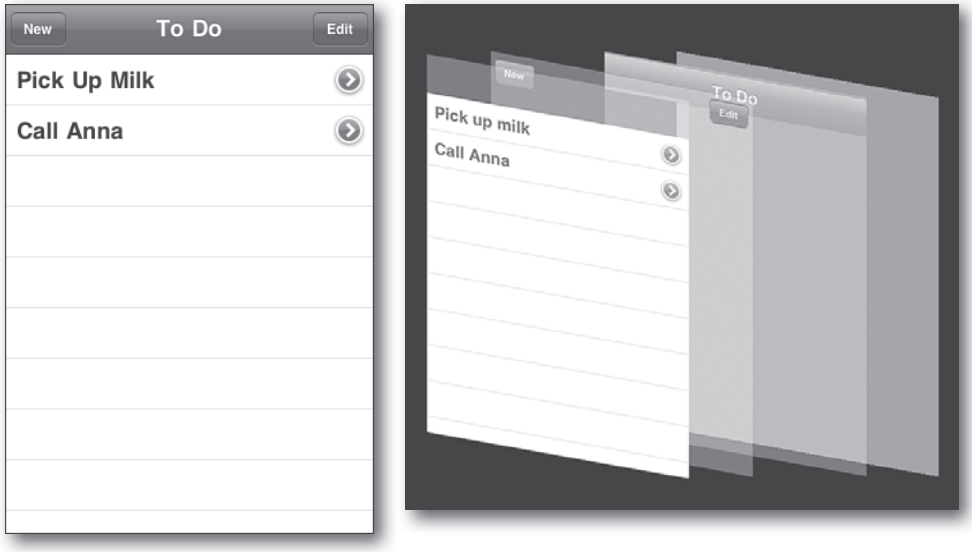
3.1 Ansichtshierarchien

Was Sie auf dem Bildschirm eines iOS-Geräts sehen, basiert auf einer baumartigen Hierarchie. Ansichten werden auf eine bestimmte hierarchische Art und Weise angeordnet, wobei das Hauptfenster den Ausgangspunkt darstellt. Alle Ansichten können Kinder, sogenannte Unteransichten, haben. Jede Ansicht, das Fenster eingeschlossen, hat eine geordnete Liste dieser Unteransichten. Ansichten können viele Unteransichten oder auch keine besitzen. Die Anwendung bestimmt, wie Ansichten angeordnet werden und welche Unteransichten zu welcher Ansicht gehören.

Unteransichten erscheinen in ihrer Reihenfolge auf dem Bildschirm, immer von hinten nach vorne. Dies entspricht der Funktionsweise der transparenten Folien bei einem Zeichentrickfilm. Es sind nur

die Teile der Folien sichtbar, die gefärbt sind. Die freien Flächen ermöglichen die Sicht auf die Elemente hinter dieser Folie. Auch Ansichten können über leere und gefüllte Bereiche verfügen und lassen sich daher zu komplexen Darstellungen überlagern.

Abbildung 3.1 zeigt ein Beispiel für die Schichten in einem typischen Fenster. Hier besitzt das Fenster eine Hierarchie, die auf einem `UINavigationController` basiert. Die Elemente sind übereinandergelagert. Das Fenster (dargestellt durch das durchsichtige Element auf der rechten Seite) hat eine Navigationsleiste, die wiederum eine Unteransicht mit zwei Schaltflächen (eine links und eine rechts) besitzt. Außerdem verfügt das Fenster über eine Tabelle mit einer eigenen Unteransicht. Diese Elemente werden übereinandergelegt und bilden so die grafische Benutzeroberfläche.



► *Abbildung 3.1: Hierarchisch gegliederte Unteransichten fügen sich zu komplexen grafischen Benutzerschnittstellen zusammen.*

Listing 3.1 zeigt die tatsächliche Ansichtshierarchie des Fensters aus *Abbildung 3.1*. Der Baum beginnt am oberen `UIWindow` und umfasst die Klassen für die einzelnen Kindansichten. Wenn Sie den Baum nach unten durchlaufen, stoßen Sie auf die Navigationsleiste (in Ebene 2) mit ihren beiden Schaltflächen (beide auf Ebene 3) sowie auf die Tabellenansicht (Ebene 4) mit ihren beiden Zeilen (jeweils Ebene 5). Einige der Elemente in der Liste sind private Klassen, die das SDK automatisch beim Stapeln von Ansichten hinzufügt. Beispielsweise wird `UILayoutContainerView` niemals direkt vom Entwickler verwendet, sondern gehört zur Implementierung des `UIWindow` durch das SDK.

Das Einzige, was in diesem Listing fehlt, sind etwa ein Dutzend Zeilentrenner in der Tabelle, die ich hier aus Platzgründen weggelassen habe. Bei diesen Zeilentrennern handelt es sich jeweils um Instanzen von `UITableViewSeparatorView`. Sie gehören zur `UITableView` und werden normalerweise auf Ebene 5 angezeigt.

```
--[ 1] UILayoutContainerView
----[ 2] UINavigationTransitionView
------[ 3] UIViewControllerWrapperView
--------[ 4] UITableView
----------[ 5] UITableViewCell
------------[ 6] UITableViewCellContentView
--------------[ 7] UILabel
------------[ 6] UIButton
--------------[ 7] UIImageView
------------[ 6] UIView
----------[ 5] UITableViewCell
------------[ 6] UITableViewCellContentView
--------------[ 7] UILabel
------------[ 6] UIButton
--------------[ 7] UIImageView
------------[ 6] UIView
----------[ 5] UIImageView
----------[ 5] UIImageView
----[ 2] UINavigationBar
------[ 3] UINavigationBarBackground
------[ 3] UINavigationItemView
------[ 3] UINavigationButton
--------[ 4] UIImageView
--------[ 4] UIButtonLabel
------[ 3] UINavigationButton
--------[ 4] UIImageView
--------[ 4] UIButtonLabel
```

▶ *Listing 3.1: Ansichtshierarchie der Aufgabenliste aus Abbildung 3.1*

3.2 Rezept: Die Ansichtshierarchie abrufen

Jede Ansicht kennt sowohl ihre Elternansicht (`[aView superview]`) als auch ihre Kindansichten (`[aView subviews]`). Um einen Ansichtsbaum wie in *Listing 3.1* zu erstellen, durchlaufen Sie rekursiv die Unteransichten einer Ansicht. Genau dies erledigt *Rezept 3.1*. Der Code baut eine grafische Baumdarstellung in der Konsole von Xcode auf, indem er sich die Klasse jeder einzelnen Ansicht merkt und bei jedem Übergang von einer Elternansicht zu deren Kindern die Einrückung verstärkt. Die Ergebnisse werden in einem veränderbaren String gespeichert und von der aufrufenden Methode zurückgegeben.

Der Baum in *Listing 3.1* wurde mit dem Code aus *Rezept 3.1* erstellt. Die Schnittstelle und das Rezept finden Sie auch im Beispielcode zu diesem Buch. Sie können diese Routine einsetzen, um das Ergebnis von *Listing 3.1* nachzuvollziehen, Sie können sie aber auch bei anderen Anwendungen einsetzen, um deren Hierarchien anzuzeigen.

```
// Rekursiver Durchlauf abwärts durch den Ansichtsbaum, wobei die
// Einrückungsebene für die Kinder jeweils erhöht wird
- (void) dumpView: (UIView *) aView atIndent: (int) indent
      into:(NSMutableString *) outstring
{
   for (int i = 0; i < indent; i++) [outstring appendString:@"—"];
   [outstring appendFormat:@"[%2d] %@\n", indent,
      [[aView class] description]];
   for (UIView *view in [aView subviews])
      [self dumpView:view atIndent:indent + 1 into:outstring];
}

// Baumrekursion beginnt bei der Wurzelansicht auf Ebene 0
- (NSString *) displayViews: (UIView *) aView
{
   NSMutableString *outstring = [[NSMutableString alloc] init];
   [self dumpView:aView atIndent:0 into:outstring];
   return outstring;
}
```

► *Rezept 3.1: Einen Ansichtsbaum durchlaufen*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.3 Rezept: Unteransichten abfragen

In Ansichten sind Arrays ihrer Kinder gespeichert, die Sie mit dem Aufruf `[aView subviews]` abrufen können. Auf dem Bildschirm werden die Kindansichten immer nach ihren Elternansichten gezeichnet, und zwar in der Reihenfolge, in der sie im Array der Unteransichten auftreten. Diese Reihenfolge spiegelt die Tatsache wider, dass die Unteransichten von hinten nach vorn gezeichnet werden. Ansichten, die weiter hinten im Array erscheinen, werden nach den Ansichten an einer früheren Position gezeichnet.

Die Methode `subviews` gibt nur die unmittelbaren Kindansichten einer gegebenen Ansicht zurück. Manchmal ist es jedoch sinnvoll, eine ausführlichere Liste der Unteransichten zu gewinnen, die auch die Enkelansichten einschließt. *Rezept 3.2* stellt die rekursive Funktion `allSubviews()` vor, die sämtliche Abkömmlinge einer Ansicht zurückgibt. Wenn Sie diese Funktion mit dem Fenster einer Sicht (über `view.window`) aufrufen, erhalten Sie die vollständige Menge der Unteransichten in dem `UIWindow`, das die Ansicht beherbergt. Diese Liste ist sehr nützlich, wenn Sie nach einer bestimmten Ansicht suchen müssen, z. B. nach einem einzelnen Schieberegler oder einer Schaltfläche.

3.3 Rezept: Unteransichten abfragen

Es ist zwar nicht üblich, doch können iOS-Anwendungen mehrere Fenster mit jeweils vielen Ansichten aufweisen. Einige davon können auf einem externen Bildschirm angezeigt werden. Eine umfassende Liste aller Anwendungsansichten erhalten Sie, indem Sie eine Iteration durch sämtliche verfügbaren Fenster durchführen. So geht die Funktion `allApplicationSubviews()` in *Rezept 3.2* vor. Ein Aufruf von `[[UIApplication sharedApplication]` gibt das Array der Anwendungsfenster zurück. Diese Funktion iteriert durch diese Fenster und fügt deren Unteransichten zu der Sammlung hinzu.

Eine Ansicht kennt nicht nur ihre Unteransichten, sondern weiß auch, zu welchem Fenster sie gehört, denn darauf zeigt ihre Eigenschaft `window`. *Rezept 3.2* enthält auch eine einfache Funktion namens `pathToView()`, die ein Array der übergeordneten Ansichten zurückgibt, und zwar vom Fenster bis zur fraglichen Ansicht. Hierzu ruft die Funktion wiederholt `superview` auf, bis sie das Fenster erreicht.

Die Vorfahren von Ansichten lassen sich auch auf andere Weise bestimmen. Die Methode `isDescendantOfView:` bestimmt, ob sich eine Ansicht innerhalb einer anderen befindet, selbst wenn diese keine direkt übergeordnete Ansicht ist. Als Ergebnis gibt die Methode einen einfachen booleschen Wert zurück. `YES` bedeutet, dass die Ansicht von der als Parameter übergebenen Ansicht abstammt.

```
// Gibt eine ausführliche Liste der Unteransichten einer Ansicht zurück
NSArray *allSubviews(UIView *aView)
{
   NSArray *results = [aView subviews];
   for (UIView *eachView in [aView subviews])
   {
      NSArray *subviews = allSubviews(eachView);
      if (subviews)
         results = [results arrayByAddingObjectsFromArray: subviews];
   }
   return results;
}

// Gibt alle Ansichten innerhalb der Anwendung zurück
NSArray *allApplicationViews()
{
   NSArray *results = [[UIApplication sharedApplication] windows];
   for (UIWindow *window in [[UIApplication sharedApplication]
      windows])
   {
      NSArray *subviews = allSubviews(window);
      if (subviews) results =
         [results arrayByAddingObjectsFromArray: subviews];
   }
   return results;
}
```

```
// Gibt ein Array der Elternansichten vom Fenster bis zur betreffenden
// Ansicht zurück.
NSArray *pathToView(UIView *aView)
{
   NSMutableArray *array = [NSMutableArray arrayWithObject:aView];
   UIView *view = aView;
   UIWindow *window = aView.window;
   while (view != window)
   {
      view = [view superview];
      [array insertObject:view atIndex:0];
   }
   return array;
}
```

▶ *Rezept 3.2: Hilfsfunktionen für Unteransichten*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.4 Unteransichten verwalten

Die Klasse `UIView` enthält viele Methoden zum Erstellen und Verwalten von Ansichten. Damit können Sie Ansichten zur Hierarchie hinzufügen und aus ihr entfernen sowie die Hierarchie umordnen und abfragen. Da die Hierarchie festlegt, was auf dem Bildschirm zu sehen ist, ändert sich bei einer Anpassung der Beziehungen zwischen den Ansichten die Darstellung auf dem iOS-Gerät. Im Folgenden finden Sie einige Verfahrensweisen für typische Aufgaben der Ansichtsverwaltung.

3.4.1 Unteransichten hinzufügen

Um einer Ansicht neue Unteransichten hinzuzufügen, rufen Sie `[parentView addSubview:child]` auf. Neue Unteransichten erscheinen stets vorn auf dem Bildschirm, da iOS sie über allen bisherigen Ansichten hinzufügt. Um eine Unteransicht an einer bestimmten Stelle der Ansichtshierarchie einfügen zu können, bietet das SDK die folgenden drei Hilfsmethoden an:

- `insertSubview: atIndex:`
- `insertSubview: aboveSubview:`
- `insertSubview: belowSubview:`

Diese Methoden legen fest, wo die neue Ansicht landet. Die Einfügung kann relativ zu einer anderen Ansicht oder an einem bestimmten Index des Arrays der Unteransichten erfolgen. Die `above-` und

`below`-Methoden fügen Unteransichten vor bzw. hinter einer gegebenen Kindansicht ein. Dabei werden vorhandene Ansichten nicht überschrieben, sondern nach vorn geschoben.

3.4.2 Unteransichten umordnen und entfernen

Bei der Benutzerinteraktion mit dem Bildschirm müssen Anwendungen häufig Ansichten umordnen und entfernen. Dazu gibt es im iOS SDK verschiedene Möglichkeiten, um die Reihenfolge und den Inhalt von Ansichten zu ändern.

- Mit `[parentView exchangeSubviewAtIndex:i withSubviewAtIndex:j]` tauschen Sie die Positionen zweier Ansichten aus.
- Um Unteransichten in den Vordergrund oder Hintergrund zu verlegen, verwenden Sie `bringSubviewToFront:` bzw. `sendSubviewToBack`.
- Um eine Unteransicht zu entfernen, rufen Sie `[childView removeFromSuperview]` auf. Wird die Kindansicht zurzeit auf dem Bildschirm angezeigt, so verschwindet sie. Wenn Sie eine Unteransicht entfernen, erhält sie eine `release`-Nachricht, sodass ihr Speicher freigegeben werden kann, falls der Beibehaltungszähler auf null gefallen ist.

Beim Umordnen, Hinzufügen und Entfernen von Ansichten wird der Bildschirm automatisch neu gezeichnet, um die neue Ansichtsdarstellung zu zeigen.

3.4.3 Ansichtscallbacks

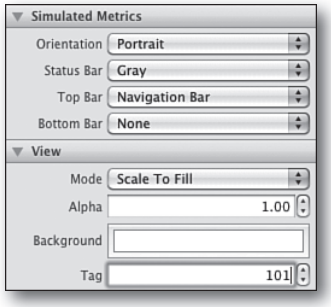
Bei einer Änderung der Ansichtshierarchie können Callbacks an die fraglichen Ansichten gesendet werden. Das iOS SDK enthält sechs solcher Callback-Methoden, mit denen sich die Anwendung auf dem neuesten Stand über Ansichten halten kann, die verschoben werden oder ihre Elternansichten ändern.

- `didAddSubview:` wird nach dem erfolgreichen Aufruf von `addSubview:` an eine Ansicht gesendet. Dadurch können Unterklassen von `UIView` beim Hinzufügen von neuen Ansichten zusätzliche Vorgänge ausführen.
- `didMoveToSuperview:` informiert Ansichten darüber, dass sie einer neuen übergeordneten Ansicht zugeordnet wurden, damit die Ansicht auf irgendeine Weise auf ihre neue Elternansicht reagieren kann. Wenn die Ansicht von ihrer übergeordneten Ansicht entfernt wurde, ist die neue Elternansicht `nil`.
- `willMoveToSuperview:` wird gesendet, bevor eine Ansicht zu einer anderen Elternansicht verschoben wird.
- `didMoveToWindow:` ist das Gegenstück zu `didMoveToSuperview:` für den Fall, dass die Ansicht nicht zu einer neuen übergeordneten Ansicht, sondern zu einer neuen `Window`-Hierarchie verschoben wird.
- `willMoveToWindow:` wird wiederum gesendet, bevor die Verschiebung auftritt.
- `willRemoveSubview:` informiert eine Ansicht darüber, dass ihre Kindansicht entfernt wird.

Ich verwende diese Methoden nur selten. Wenn ich es tue, dann erweisen Sie sich jedoch fast immer als Retter in der Not, da ich mit ihrer Hilfe Verhalten hinzufügen kann, ohne im Voraus zu wissen, welche Art von Unter- oder Oberansichtsklasse verwendet wird. Die Fenstercallbacks sind hauptsächlich dazu da, Überlagerungsansichten in einem zweiten Fenster anzuzeigen, z. B. Warnungen oder Eingabeansichten wie Tastaturen.

3.5 Rezept: Ansichten mit Tags versehen und abrufen

Das iOS SDK verfügt über eine integrierte Suchfunktion, mit der Sie Ansichten abrufen können, indem Sie sie mit Tags versehen. Tags sind einfach nur Zahlen, gewöhnlich positive Ganzzahlen, die eine Ansicht bezeichnen. Zugewiesen werden sie über die Eigenschaft `tag` einer Ansicht, z. B. als `myView.tag = 101`. Im Interface Builder können Sie das Tag einer Ansicht im Attribut-Informationsfeld festlegen. Wie Sie in *Abbildung 3.2* sehen, geben Sie das Tag im Abschnitt **View** an.



▶ *Abbildung 3.2: Das Tag für eine Ansicht legen Sie im Attribut-Informationsfenster des Interface Builder fest.*

Tags können Sie völlig willkürlich wählen, reserviert ist nur die 0 als Standardeinstellung für alle neu erstellten Ansichten. Die Entscheidung, wie die Tags angewendet werden und welche Werte sie annehmen, liegt ganz bei Ihnen. Sie können jegliche Instanzen mit Tags versehen, die Kinder von `UIView` sind, auch Fenster und Steuerelemente. Befinden sich auf dem Bildschirm also viele Schaltflächen und Umschalter, können Sie sie mithilfe von Tags unterscheiden, wenn die Benutzer sie auslösen. Fügen Sie zu Ihren Callback-Methoden eine einfache `switch`-Anweisung hinzu, damit sich die Methode das Tag ansieht und daraus ableitet, wie sie reagieren muss.

Apple selbst verwendet Tags für Unteransichten nur selten. Die einzigen Fälle, die ich bisher gesehen habe, sind die Schaltflächen in `UIAlertView`, die mit den Tags 1, 2 usw. bezeichnet sind. (Ich bin fast überzeugt davon, dass diese Tags nur aus Versehen dort zurückgeblieben sind.) Wenn Sie sich Sorgen über Konflikte mit Apple-Tags machen, beginnen Sie mit der Nummerierung bei 10 oder 100 oder irgendeiner anderen Zahl, die höher ist als die Werte, die Apple wahrscheinlich benutzt.

3.5.1 Tags zur Suche nach Ansichten verwenden

Mit Tags können Sie es vermeiden, Elemente der Benutzerschnittstelle in Ihrem Programm übergeben zu müssen, da Sie sie direkt von irgendeiner Elternansicht aus zugänglich machen. Die Methode `viewWithTag:` ruft eine mit Tags versehene Ansicht von der tiefer gelegenen Hierarchie ab. Die Suche erfolgt rekursiv, sodass das mit einem Tag versehene Element kein unmittelbares Kind der

fraglichen Ansicht sein muss. Es ist möglich, dass Sie die Suche mit `[window viewWithTag:101]` vom Fenster aus durchführen und dabei eine Ansicht finden, die sich mehrere Äste tiefer im Hierarchiebaum befindet. Wenn ein Tag von mehreren Ansichten verwendet wird, gibt die Methode `viewWithTag:` das erste dieser Elemente zurück, das sie findet.

Das Problem bei `viewWithTag:` ist, dass diese Methode ein `UIView`-Objekt zurückgibt, sodass Sie es erst in den richtigen Typ umwandeln müssen, um es verwenden zu können. Nehmen wir an, dass Sie eine Beschriftung (`UIlabel`) abrufen und deren Text festlegen möchten.

```
UILabel *label = (UILabel *)[self.view.window viewWithTag:101];
label.text = @"Hello World";
```

Es wäre weit einfacher, einen Aufruf zu verwenden, der ein bereits typisiertes Objekt zurückgibt, sodass Sie es unmittelbar verwenden können. Dies geschieht in den folgenden Aufrufen:

```
- (IBAction)updateTime:(id)sender
{
   // Setzt die Beschriftung auf die aktuelle Uhrzeit
   [self.view.window labelWithTag:LABEL_TAG].text =
      [[NSDate date] description];
}

- (IBAction)updateSwitch:(id)sender
{
   // Schaltet den Umschalter von der derzeitigen Einstellung um
   UISwitch *s = [self.view.window switchWithTag:SWITCH_TAG];
   [s setOn:!s.isOn];
}
```

Rezept 3.3 erweitert das Verhalten von `UIView`, um die neue Kategorie `TagExtensions`. Diese Kategorie fügt nur zwei typisierte Tag-Methoden für `UILabel` und `UISwitch` hinzu. Im Beispielcode zu diesem Buch ist dies zu einer vollständigen Suite von typisierten Tag-Hilfsmethoden ausgebaut. Aus Platzgründen habe ich hier auf die anderen Klassen verzichtet, doch sie folgen demselben Muster der Typumwandlung von `viewWithTag:`. Zugriff auf die vollständige Sammlung erhalten Sie, wenn Sie die `UIView-TagExtensions`-Dateien in Ihre Projekte aufnehmen.

Hinweis

Tags sind auch noch zu etwas anderem nützlich: Bei der Arbeit mit Tabellenzellen in Tabellenansichten können Sie die Schaltflächen mit einem Tag versehen, das die jeweilige Zeile im `indexPath` angibt. Dadurch können Sie die Zelle abrufen, wenn der Benutzer auf die Zelle tippt.

```
@interface UIView (TagExtensions)
- (UILabel *) labelWithTag: (NSInteger) aTag;
- (UISwitch *) switchWithTag: (NSInteger) aTag;
@end
```

```
@implementation UIView (TagExtensions)
- (UILabel *) labelWithTag: (NSInteger) aTag
{
   UIView *aView = [self viewWithTag:aTag];
   if (aView && [aView isKindOfClass:[UILabel class]])
      return (UILabel *) aView;
   return nil;
}

- (UISwitch *) switchWithTag: (NSInteger) aTag
{
   UIView *aView = [self viewWithTag:aTag];
   if (aView && [aView isKindOfClass:[UISwitch class]])
      return (UISwitch *) aView;
   return nil;
}
@end
```

▶ *Rezept 3.3: Mit Tags gekennzeichnete Ansichten mit korrekt umgewandelten Objekten abrufen*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.6 Rezept: Ansichten benennen

Die Verwendung von Tags ist zwar eine solide Vorgehensweise zur Bezeichnung von Ansichten, doch manche Entwickler arbeiten lieber mit Namen als mit Zahlen. Dadurch erhält das Bezeichnungsschema für die Ansichten noch eine zusätzliche Bedeutung. Anstatt von der »Sicht mit dem Tag 101« sprechen Sie von einem Umschalter mit dem Namen »Ignition Switch« (»Auslöser«), was dessen Rolle beschreibt und für eine gewisse Form von Selbstdokumentation sorgt, die eine reine Zahl nicht bietet.

```
// Umschalter betätigen
UISwitch *s = [self.view switchNamed:@"Ignition Switch"];
[s setOn:!s.isOn];
```

Es ist relativ einfach, eine Klasse zu entwerfen, die Strings zu Ansichts-Tags zuordnet. Dazu gibt es zwei Möglichkeiten. Erstens können Sie eine benutzerdefinierte Klasse schreiben, die ein Dictionary speichert. Darin werden die Namen mit Tags verknüpft, sodass Ansichten solche Namen registrieren und die Registrierung wieder aufheben können. Die zweite Möglichkeit besteht darin, die zur Laufzeit verknüpften Objektfunktionen von Objective-C zu verwenden.

3.6.1 Verknüpfte Objekte

Verknüpfte Objekte lassen sich zwar sauberer und einfacher verwenden, werden in iOS 5 aber kaum unterstützt. Zurzeit müssen Sie bei der Verlinkung mit dem Foundation-Framework sowohl Konstanten als auch Funktionen manuell deklarieren. Wenn Sie vor einer solchen codeintensiven Arbeit zurückschrecken, sollten Sie lieber das Rezept im folgenden Abschnitt einsetzen, bei dem Dictionarys verwendet werden.

```
enum {
   OBJC_ASSOCIATION_ASSIGN = 0,
   OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1,
   OBJC_ASSOCIATION_COPY_NONATOMIC = 3,
   OBJC_ASSOCIATION_RETAIN = 01401,
   OBJC_ASSOCIATION_COPY = 01403
};

typedef uintptr_t objc_AssociationPolicy;
id objc_getAssociatedObject(id object, void *key);
void objc_setAssociatedObject(id object, void *key, id value,
   objc_AssociationPolicy policy);
void objc_removeAssociatedObjects(id object);
```

Um die verknüpften Objekte nutzen zu können, deklarieren Sie die eigene Eigenschaft `nametag`. Sie erstellt keinen neuen Speicher in einer `UIView`, sondern fügt zwei Methoden hinzu (eine Set- und eine Get-Methode), die die Laufzeitfunktionen nutzen.

```
@interface UIView (Nametags)
@property (nonatomic, strong) NSString *nametag;
- (UIView *) viewWithNametag: (NSString *) aName;
@end
```

Die Implementierung ist einfach. Um die verknüpfte Eigenschaft `nametag` festzulegen und abzurufen, müssen Sie zwei Laufzeitfunktionen in Objective-C aufrufen:

```
static const char *NametagKey = "Nametag Key";

- (id) nametag
{
   return objc_getAssociatedObject(self, (void *) NametagKey);
}

- (void)setNametag:(NSString *) theNametag
{
   objc_setAssociatedObject(self, (void *) NametagKey,
   theNametag, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}
```

Um eine Ansicht anhand ihrer `nametag`-Eigenschaft abzurufen, führen Sie eine einfache Tiefensuche durch. Die im Folgenden gezeigte Methode `viewWithNameTag:` gibt die erste Ansicht zurück, deren Tag einem gegebenen String entspricht. Wird keine übereinstimmende Ansicht gefunden, gibt die Methode `nil` zurück.

```
- (UIView *) viewWithNametag: (NSString *) aName
{
   if (!aName) return nil;

   // Ist dies die richtige Ansicht?
   if ([self.nametag isEqualToString:aName])
      return self;

   // Rekursive Tiefensuche in den Unteransichten
   for (UIView *subview in self.subviews)
   {
      UIView *resultView = [subview viewNamed:aName];
      if (resultView) return resultView;
   }

   // Nicht gefunden
   return nil;
}
```

3.6.2 Ein Namens-Dictionary verwenden

Rezept 3.4 zeigt, wie Sie einen Ansichtsnamen-Manager erstellen können. Hier wird eine Singleton-Instanz genutzt (`[ViewIndexer sharedInstance]`), um das Tag/Namen-Dictionary zu speichern.

Die Klasse verlangt eindeutige Namen. Wenn ein Ansichtsname bereits registriert ist, schlägt eine weitere Registrierungsanforderung fehl. Wurde eine Ansicht bereits unter einem anderen Namen registriert, hebt die zweite Registrierung die erste auf. Es gibt natürlich einige Möglichkeiten, um das zu umgehen. Wenn Sie die Tags einer Ansicht ändern und sie dann erneut registrieren, kann der Indizierer nicht mehr erkennen, dass er diese Ansicht bereits registriert hat. Wenn Sie also diesen Ansatz verwenden, sollten Sie die Tags im Interface Builder angeben oder automatisch vom Registrierungsprozess festlegen lassen und die Tags ansonsten nicht mehr anfassen.

Falls Sie die Ansichten manuell erstellen, sollten Sie sie im selben Schritt registrieren und zur Ansichtshierarchie hinzufügen. Bei einer im Interface Builder erstellten Ansicht registrieren Sie die Namen in `viewDidLoad` mit den Tag-Nummern, die Sie im Attribute-Informationsfeld festgelegt haben.

```
- (void) viewDidLoad
{
   [[self.view viewWithTag:LABEL_TAG] registerName:@"my label"];
   [[self.view viewWithTag:SWITCH_TAG] registerName:@"my switch"];
}
```

Rezept 3.4 verbirgt die Klasse für den Ansichtsindizierer vor der Öffentlichkeit. Der Code verpackt die Aufrufe in einer `UIView`-Kategorie für Namenserweiterungen. Aus Platzgründen werden in dem Rezept keine Namensabrufe wie `labelNamed:` und `textFieldNamed:` gezeigt, aber sie sind im Beispielcode zu diesem Kapitel enthalten.

```
@interface ViewIndexer : NSObject {
   NSMutableDictionary *tagdict;
   NSInteger count;
}
@end

@implementation ViewIndexer
static ViewIndexer *sharedInstance = nil;

+(ViewIndexer *) sharedInstance {
   if(!sharedInstance) sharedInstance = [[self alloc] init];
   return sharedInstance;
}

- (id) init
{
   if (!(self = [super init])) return self;
   tagdict = [NSMutableDictionary dictionary];
   count = 10000;
   return self;
}

// Generiert eine neue Nummer und erhöht den Zählerwert
- (NSInteger) pullNumber
{
   return count++;
}

// Prüft, ob der Name bereits im Dictionary vorhanden ist
- (BOOL) nameExists: (NSString *) aName
{
   return [tagdict objectForKey:aName] != nil;
}

// Ruft den ersten übereinstimmenden Namen für das Tag ab
- (NSString *) nameForTag: (NSInteger) aTag
{
   NSNumber *tag = [NSNumber numberWithInt:aTag];
   NSArray *names = [tagdict allKeysForObject:tag];
```

```
    if (!names) return nil;
    if ([names count] == 0) return nil;
    return [names objectAtIndex:0];
}

// Gibt das Tag für einen registrierten Namen zurück. Wird keines gefunden,
// erfolgt die Rückgabe von 0.
- (NSInteger) tagForName: (NSString *)aName
{
    NSNumber *tag = [tagdict objectForKey:aName];
    if (!tag) return 0;
    return [tag intValue];
}

// Aufheben der Registrierung setzt das Tag auf 0 zurück
- (BOOL) unregisterName: (NSString *) aName forView: (UIView *) aView
{
    NSNumber *tag = [tagdict objectForKey:aName];

    // Tag nicht gefunden
    if (!tag) return NO;

    // Tag stimmt nicht mit dem registrierten Namen überein
    if (aView.tag != [tag intValue]) return NO;

    aView.tag = 0;
    [tagdict removeObjectForKey:aName];
    return YES;
}

// Registriert einen neuen Namen. Namen können nicht zweimal registriert
// werden. (Zuerst muss die erste Registrierung aufgehoben werden.) Ist
// eine Ansicht bereits registriert, wird die erste Registrierung
// aufgehoben und dann die zweite durchgeführt.
- (NSInteger) registerName:(NSString *)aName forView: (UIView *) aView
{
    // Sie können einen vorhandenen Namen nicht erneut registrieren
    if ([[ViewIndexer sharedInstance] nameExists:aName]) return 0;

    // Prüft, ob die Ansicht bereits benannt ist. Wenn ja, wird ihre
    // Registrierung aufgehoben.
    NSString *currentName = [self nameForTag:aView.tag];
    if (currentName) [self unregisterName:currentName forView:aView];
```

```
    // Registriert das vorhandene Tag oder generiert ein neues,
    // wenn aView.tag = 0 ist.

    if (!aView.tag) aView.tag = [[ViewIndexer sharedInstance]
       pullNumber];
    [tagdict
       setObject:[NSNumber numberWithInt:aView.tag]
       forKey: aName];
    return aView.tag;
}
@end

@implementation UIView (NameExtensions)
- (NSInteger) registerName: (NSString *) aName
{
    return [[ViewIndexer sharedInstance] registerName: aName
       forView: self];
}

- (BOOL) unregisterName: (NSString *) aName
{
    return [[ViewIndexer sharedInstance] unregisterName: aName
       forView:self];
}

- (UIView *) viewNamed: (NSString *) aName
{
    NSInteger tag = [[ViewIndexer sharedInstance] tagForName: aName];
    return [self viewWithTag: tag];
}
@end
```

▶ *Rezept 3.4: Einen Ansichtsnamen-Manager erstellen*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.7 Geometrie von Ansichten

Wie zu erwarten ist, spielt die Geometrie bei der Arbeit mit Ansichten eine wichtige Rolle. Sie definiert, wo die einzelnen Ansichten auf dem Bildschirm erscheinen, wie groß sie sind und welche Ausrichtung sie haben. Die Klasse `UIView` enthält zwei Eigenschaften zur Definition dieser Aspekte.

Jede Ansicht definiert ihre Ausdehnung mit einem Rahmen, der ihre Abgrenzungen angibt: die Position, die Breite und die Höhe. Wenn Sie den Rahmen einer Ansicht ändern, wird sie aktualisiert, damit sie in den neuen Rahmen passt. Bei einer Vergrößerung der Breite wird die Ansicht gestreckt, bei einer neuen Position verschoben. Der Rahmen legt die Grenzen der Ansicht auf dem Bildschirm fest. Eine Ansicht muss nicht unbedingt die gleiche Größe wie der Bildschirm aufweisen, sondern kann größer oder kleiner sein. Es ist auch möglich, dass eine Ansicht größer oder kleiner als ihre Elternansicht ist.

Ansichten haben auch die Eigenschaft `transform`, die ihre Ausrichtung und jegliche darauf angewandten geometrischen Transformationen festlegt. So können Sie eine Ansicht beispielsweise dehnen oder strecken, indem Sie eine Transformation auf sie anwenden, oder sie aus der Vertikale kippen. Rahmen und Transformationen definieren zusammen die Geometrie einer Ansicht.

3.7.1 Rahmen

Für die Rechtecke der Rahmen wird eine `CGRect`-Struktur verwendet, die – wie das Präfix `CG` schon andeutet – im Core Graphics-Framework definiert ist. Ein `CGRect` besteht aus einem Ursprung (einem `CGPoint`, `x` und `y`) und einer Größe (einer `CGSize`, Breite und Höhe). Wenn Sie Ansichten erstellen, weisen Sie sie gewöhnlich zu und initialisieren sie wie im folgenden Beispiel mit einem Rahmen:

```
CGRect rect = CGRectMake(0.0f, 0.0f, 320.0f, 416.0f);
myView = [[UIView alloc] initWithFrame: rect];
```

Die Funktion `CGRectMake` erstellt ein neues Rechteck mit vier Parametern, nämlich den `x`- und `y`-Koordinaten des Ursprungs sowie der Breite und Höhe des Rechtecks. Neben `CGRectMake` gibt es noch mehrere andere Hilfsfunktionen, die Sie bei der Arbeit mit Rechtecken und Rahmen unterstützen.

- `NSStringFromCGRect(aCGRect)` konvertiert eine `CGRect`-Struktur in einen formatierten String. Diese Funktion macht es einfach, den Rahmen einer Ansicht beim Debugging zu protokollieren.
- `CGRectFromString(aString)` stellt ein Rechteck aus seiner Stringdarstellung wieder her. Dies ist nützlich, wenn Sie den Rahmen einer Ansicht als String in den Benutzervoreinstellungen gespeichert haben und diesen String wieder in ein `CGRect` zurückverwandeln wollen.
- Mit `CGRectInset(aRect, xinset, yinset)` können Sie ein kleineres oder größeres Rechteck mit demselben Zentrum wie das ursprüngliche erstellen. Verwenden Sie für kleinere Rechtecke einen positiven und für größere einen negativen `inset`-Wert.

- `CGRectOffset(aRect, xoffset, yoffset)` gibt ein Rechteck zurück, das um die von Ihnen festgelegten `x`- und `y`-Werte gegenüber dem ursprünglichen Rechteck versetzt ist. Das ist nützlich, wenn Sie Rahmen auf dem Bildschirm verschieben oder mithilfe der Unterebenen einer Ansicht Schlagschatten gestalten möchten.
- `CGRectGetMidX(aRect)` und `CGRectGetMidY(aRect)` rufen die `x`- und `y`-Koordinaten ab, die der Mittelpunkt des Rechtecks aufweist. Mit diesen Funktionen lassen sich die Mittelpunkte von Grenzen und Rahmen auf einfache Weise ermitteln.
- Mithilfe von `CGRectIntersectsRect(rect1, rect2)` können Sie erfahren, ob sich zwei Rechtecke schneiden. Verwenden Sie diese Funktion, wenn Sie wissen möchten, ob sich zwei Rechtecke auf dem Bildschirm überlappen.
- `CGRectZero` ist eine Konstante für ein Rechteck, das sich bei `(0,0)` befindet und dessen Breite und Höhe null betragen. Sie können diese Konstante verwenden, wenn Sie einen Rahmen erstellen müssen, aber noch nicht wissen, welche Größe und welche Position er zum Erstellungszeitpunkt haben wird.

Die `CGRect`-Struktur besteht aus zwei Unterstrukturen: `CGPoint` definiert den Ursprung des Rechtecks, `CGSize` seine Ausdehnung. Punkte sind Positionen, die in `x`- und `y`-Koordinaten definiert sind, Größen setzen sich aus Breite und Höhe zusammen. Mit `CGPointMake(x, y)` erstellen Sie Punkte, mit `CGSizeMake(width, height)` legen Sie Größen fest. Diese beiden Strukturen wirken zwar gleichartig (beide weisen je zwei Fließkommawerte auf), werden vom iOS SDK aber unterschieden. Punkte sind Positionen, Größen sind Ausdehnungen. Sie können `myFrame.size` nicht auf einen Punkt setzen.

Wie Rechtecke können Sie auch Punkte und Größen in Strings umwandeln und umgekehrt. Dazu dienen die Funktionen `NSStringFromCGPoint()`, `NSStringFromCGSize()`, `CGSizeFromString()` und `CGPointFromString()`. Es ist auch möglich, Punkte und Größen in Dictionarys zu verwandeln und umgekehrt.

3.7.2 Transformationen

Im Rahmen seiner Core Graphics-Implementierung unterstützt das iOS SDK standardmäßige affine Transformationen. Damit können Sie Punkte aus einem Koordinatensystem in ein anderes umwandeln. Diese Funktionen werden sehr häufig bei 2D- und 3D-Animationen eingesetzt. Die Version im iOS SDK verwendet eine 3×3-Matrix, um `UIView`-Transformationen zu definieren, was sie zu einer reinen 2D-Lösung macht. Für 3D-Transformationen wird eine 4×4-Matrix benötigt, und dies ist der Standard für Core Animation-Ebenen. Mit affinen Transformationen können Sie Ansichten in Echtzeit in der Größe ändern, verschieben und drehen. Dazu legen Sie wie im folgenden Beispiel die Eigenschaft `transform` der Ansicht fest:

```
float angle = theta * (PI / 100);
CGAffineTransform transform = CGAffineTransformMakeRotation(angle);
myView.transform = transform;
```

Die Transformation wird stets in Bezug auf den Mittelpunkt der Ansicht durchgeführt. Wenn Sie also wie hier eine Drehung durchführen, erfolgt sie um den Mittelpunkt der Ansicht. Brauchen Sie eine Drehung um einen anderen Punkt, müssen Sie die Ansicht erst verschieben, dann drehen und wieder zurückverschieben.

Um Änderungen rückgängig zu machen, setzen Sie die Eigenschaft `transform` auf die Identitätstransformation. Dadurch wird die Ansicht auf die letzte Einstellung für den Rahmen zurückgesetzt.

```
myView.transform = CGAffineTransformIdentity;
```

Hinweis

In iOS hat die `y`-Koordinate ihren Nullpunkt oben und nach unten zu höhere Werte. Das entspricht dem Koordinatensystem in PostScript, ist dem früher auf dem Mac verwendeten Koordinatensystem von Quartz aber genau entgegengesetzt. Der Ursprung ist in iOS die obere linke Ecke, nicht die untere linke.

3.7.3 Koordinatensysteme

Ansichten sind Wanderer zwischen den Welten: Ihre Rahmen sind im Koordinatensystem ihrer Elternansicht definiert, ihre Grenzen und Unteransichten aber in ihrem eigenen. Das iOS SDK enthält verschiedene Werkzeuge, um sich zwischen diesen beiden Koordinatensystemen zu bewegen, solange die Ansicht nur im selben `UIWindow` bleibt. Um einen Punkt aus einer anderen Ansicht in das eigene Koordinatensystem zu konvertieren, verwenden Sie wie im folgenden Beispiel `convertPoint:fromView:`:

```
myPoint = [myView convertPoint:somePoint fromView:otherView];
```

Wenn der ursprüngliche Punkt die Position eines Objekts angegeben hat, so tut er das auch nach wie vor, wobei sich die Koordinaten jetzt jedoch auf den Ursprung von `myView` beziehen. Um andersherum vorzugehen, transformieren Sie mit `convertPoint:toView:` einen Punkt in das Koordinatensystem einer anderen Ansicht. Ähnlich funktionieren `convertRect:toView:` und `convertRect:fromView:`. Hierbei werden aber keine `CGPoint`-, sondern `CGRect`-Strukturen umgewandelt.

Beachten Sie, dass das Koordinatensystem eines iOS-Systems nicht unbedingt mit dem Pixelsystem zu seiner Darstellung übereinstimmen muss. Das diskrete 640×960-Pixel-Retina Display auf dem iPhone 4 und dem iPod touch der vierten Generation wird beispielsweise über ein kontinuierliches 320×480-Koordinatensystem im SDK angesprochen. Sie können zwar Grafiken höherer Qualität bereitstellen, um die Pixel auf den moderneren Geräten aufzufüllen, aber alle Positionen, die Sie im Code angeben, greifen auf dasselbe Koordinatensystem zu, das auch auf älteren Geräten mit geringerer Pixeldichte verwendet wird. Die Position `(160.0, 240.0)` bleibt auf dem iPhone und dem iPod touch ungefähr in der Mitte des Bildschirms.

3.8 Rezept: Mit Ansichtsrahmen arbeiten

Wenn Sie den Rahmen einer Ansicht ändern, aktualisieren Sie seine Größe (also seine Breite und Höhe) und seine Position. Beispielsweise können Sie einen Rahmen wie im folgenden Beispiel verschieben. Der Code erstellt eine Unteransicht bei `(0.0, 0.0)` und verschiebt sie abwärts zu `(0.0, 30.0)`.

```
CGRect initialRect = CGRectMake(0.0f, 0.0f, 320.0f, 50.0f);
myView = [[UIView alloc] initWithFrame:initialRect];
[topView addSubview:myView];
myView.frame = CGRectMake(0.0f, 30.0f, 320.0f, 50.0f);
```

Diese Vorgehensweise ist jedoch eher unüblich. Das iOS SDK erwartet nicht, dass Sie eine Ansicht verschieben, indem Sie ihren Rahmen ändern. Stattdessen haben Sie die Möglichkeit, die Position einer Ansicht zu aktualisieren. Am besten geht das, indem Sie den Mittelpunkt der Ansicht festlegen. Dazu können Sie direkt auf die integrierte Ansichtseigenschaft `center` zugreifen:

```
myView.center = CGPointMake(160.0f, 55.0f);
```

Wahrscheinlich erwarten Sie jetzt, dass das SDK Ihnen auch einen Weg bietet, um eine Ansicht durch Änderung ihres Ursprungs zu verschieben, doch eine solche Möglichkeit besteht nicht. Sie müssen den Rahmen der Ansicht abrufen, dessen Ursprung auf den gewünschten Punkt setzen und dann den Rahmen aktualisieren. Das folgende Fragment erstellt jedoch die neue Eigenschaft `origin`, mit der Sie den Ursprung der Ansicht abrufen und ändern können.

```
@interface UIView (ViewGeometry)
@property CGPoint origin;
@end


@implementation UIView (ViewGeometry)
- (CGPoint) origin
{
   return self.frame.origin;
}

- (void) setOrigin: (CGPoint) aPoint
{
   CGRect newframe = self.frame;
   newframe.origin = aPoint;
   self.frame = newframe;
}
@end
```

Da diese Erweiterung einen so offensichtlichen Eigenschaftennamen verwendet, kann der Code aufgrund eines Namenskonflikts unbrauchbar werden, wenn Apple eines Tages selbst eine solche Möglichkeit implementieren sollte. In den Beispielen in diesem Buch habe ich freimütig Gebrauch von offensichtlichen Namen gemacht. In Produktionscode sollten Sie das nicht tun. Verwenden Sie Ihre persönlichen Initialen oder die des Unternehmens als Präfix, um eigene Bezeichnungen von anderen absetzen zu können.

Wenn Sie eine Ansicht verschieben, müssen Sie sich keine Sorgen über Dinge wie rechteckige Bereiche machen, die sichtbar sind oder verdeckt werden. iOS kümmert sich darum, die Oberfläche neu zu zeichnen. Dadurch können Sie Ihre Ansichten wie greifbare Objekte behandeln und können die Probleme der Darstellung getrost Cocoa Touch überlassen.

3.8.1 Die Größe anpassen

Bei der einfachsten Verwendung wird die Größe einer Ansicht durch ihren Rahmen und ihre Grenzen bestimmt. Wie Sie bereits wissen, definiert der Rahmen die Position einer Ansicht im Koordinatensystem der Elternansicht. Liegt der Ursprung eines Rahmens also bei `(0.0, 30.0)`, so erscheint die Ansicht in der übergeordneten Ansicht bündig an der linken Seite und oben um 30 Pixel verschoben. Auf älteren Displays entspricht das tatsächlich 30 Pixeln, auf Retina Displays 60 Pixeln. Die Grenzen definieren die Ansicht in ihrem eigenen Koordinatensystem. Der Ursprung für die Grenzen einer Ansicht (also `myView.bounds`) ist also immer `(0.0, 0.0)`, und ihre Größe entspricht der normalen Ausdehnung (also der Eigenschaft `size` des Rahmens).

Die Größe einer Ansicht auf dem Bildschirm ändern Sie, indem Sie ihren Rahmen oder ihre Grenze ändern. Technisch gesprochen, aktualisieren Sie die Größenkomponente dieser Strukturen. Wie beim Verschieben von Ursprüngen ist es einfach, eine eigene Hilfsmethode zu schreiben, um die Aufgabe direkt zu erledigen.

```
- (void) setSize: (CGSize) aSize
{
   CGRect newframe = self.frame;
   newframe.size = aSize;
   self.frame = newframe;
}
```

Wenn sich die Größe einer Ansicht ändert, wird die Ansicht selbst live auf dem Bildschirm aktualisiert. Je nachdem, wie die Elemente in der Ansicht und die Klasse der Ansicht definiert sind, können Unteransichten dabei abgeschnitten oder so verkleinert werden, dass sie passen. Es gibt keine allgemeine Regel für alle möglichen Situationen. Im Größen-Informationsfenster des Interface Builder von Xcode finden Sie eine Reihe von Optionen zur Größenänderung, mit denen Sie festlegen können, wie die Unteransichten auf Änderungen des Rahmens ihrer Elternansicht reagieren. In Kapitel 1, »*Benutzeroberflächen entwerfen*«, erfahren Sie mehr darüber, wie Sie Elemente im Interface Builder gestalten.

Hinweis

Die Grenzen werden durch die Transformation einer Ansicht beeinflusst, eine mathematische Komponente, die das Erscheinungsbild der Ansicht auf dem Bildschirm ändert. Ändern Sie den Rahmen einer Ansicht nicht, wenn Sie mit Transformationen arbeiten, da Sie sonst nicht die erwarteten Ergebnisse erhalten. Beispielsweise kann es sein, dass der Ursprung des Rahmens nach einer Transformation nicht mehr mit dem Ursprung der Grenzen übereinstimmt. Die normale Reihenfolge bei der Änderung einer Ansicht besteht darin, zuerst den Rahmen oder die Grenzen festzulegen, dann den Mittelpunkt und dann ggf. die Transformationen anzuwenden.

Manchmal müssen Sie eine Ansicht in der Größe ändern, bevor Sie sie einer neuen Elternansicht hinzufügen. Stellen Sie sich z. B. eine Bildansicht vor, die Sie in einer Warnung platzieren möchten. Um die Ansicht passend zu machen, ohne ihr Seitenverhältnis zu ändern, können Sie eine Methode wie die folgende verwenden, die sicherstellt, dass Höhe und Breite proportional geändert werden.

```
- (void) fitInSize: (CGSize) aSize
{
   CGFloat scale;
   CGRect newframe = self.frame;

   if (newframe.size.height > aSize.height)
   {
      scale = aSize.height / newframe.size.height;
      newframe.size.width *= scale;
      newframe.size.height *= scale;
   }

   if (newframe.size.width >= aSize.width)
   {
      scale = aSize.width / newframe.size.width;
      newframe.size.width *= scale;
      newframe.size.height *= scale;
   }

   self.frame = newframe;
}
```

3.8.2 Die Mittelpunkte von CGRect-Objekten

Wie Sie bereits gelernt haben, verwenden `UIViews` zur Definition ihrer Rahmen `CGRect`-Strukturen, die aus einem Ursprung und einer Größe bestehen. Diese Strukturen enthalten aber keine Verweise auf den Mittelpunkt. Wenn Sie eine Ansicht an eine andere Stelle verschieben, wird zur Aktualisierung der Position aber die Eigenschaft `center` von `UIView` verwendet. Leider verwendet Core Graphics keine Mittelpunkte zur Definition von Rechtecken. Die Fähigkeiten von Core Graphics, was Mittelpunkte betrifft, beschränken sich darauf, den Mittelpunkt eines Rechtecks entlang der X- oder Y-Achse abzurufen.

Diese Lücke können Sie füllen, indem Sie eine Funktion erstellen, die zwischen den mit dem Ursprung definierten `CGRect`-Strukturen und den über den Mittelpunkt bestimmten `UIView`-Objekten vermittelt. Die folgende Funktion ruft den Mittelpunkt eines Rechtecks ab, indem sie aus den Mittelpunkten der X- und Y-Achse einen Punkt konstruiert. Sie nimmt ein Argument entgegen, nämlich das Rechteck, und gibt dessen Mittelpunkt zurück.

```
CGPoint CGRectGetCenter(CGRect rect)
{
   CGPoint pt;
   pt.x = CGRectGetMidX(rect);
   pt.y = CGRectGetMidY(rect);
   return pt;
}
```

Eine Funktion, die ein Rechteck anhand seines Mittelpunkts verschiebt und damit den Umgang mit `UIView`-Objekten nachahmt, kann ebenfalls nützlich sein. Nehmen wir an, Sie möchten eine Ansicht zu einer neuen Position verschieben, wobei sie aber innerhalb des Rahmens ihrer Elternansicht bleiben muss. Um vor der Verschiebung einen Test durchzuführen, können Sie eine Funktion wie die folgende verwenden, um den Ansichtsrahmen zu einem neuen Mittelpunkt zu verschieben. Dann können Sie den verschobenen Rahmen mit dem Elternrahmen vergleichen (mit `CGRectContainsRect()`), um sicherzustellen, dass die Ansicht nicht aus ihrem Container herausragt.

```
CGRect CGRectMoveToCenter(CGRect rect, CGPoint center)
{
   CGRect newrect = CGRectZero;
   newrect.origin.x = center.x-CGRectGetMidX(rect);
   newrect.origin.y = center.y-CGRectGetMidY(rect);
   newrect.size = rect.size;
   return newrect;
}
```

Oft müssen Sie auch eine Ansicht in einer anderen zentrieren. Mit dem folgenden Code können Sie einen Rahmen abrufen, der einem zentrierten Unterrechteck entspricht. Wenn Sie eine Unteransicht hinzufügen, übergeben Sie die Grenzen der äußeren Ansicht. (Das Koordinatensystem der Unteransicht muss bei `0, 0` beginnen.) Fügen Sie die neue Ansicht dagegen im Elternobjekt der äußeren Ansicht hinzu, übergeben Sie deren Rahmen.

```
CGRect CGRectCenteredInRect(CGRect rect, CGRect mainRect)
{
   CGFloat xOffset = CGRectGetMidX(mainRect)-CGRectGetMidX(rect);
   CGFloat yOffset = CGRectGetMidY(mainRect)-CGRectGetMidY(rect);
   return CGRectOffset(rect, xOffset, yOffset);
}
```

3.8.3 Weitere Hilfsmethoden

Wie Sie gesehen haben, ist es sehr bequem, neben dem Mittelpunkt auch den Ursprung und die Größe einer Ansicht bereitzustellen, um Core Graphics-Aufrufe in ihrer natürlichen Art und Weise durchführen zu können. Diese Idee können Sie noch weiter ausbauen und auch andere Eigenschaften einer Ansicht offenlegen, z. B. ihre Breite (`width`) und ihre Höhe (`height`) sowie grundlegende geometrische Aspekte wie die Punkte `left`, `right`, `top` und `bottom`.

In gewisser Weise hintertreiben Sie damit das Designmodell von Apple, denn dadurch legen Sie Elemente offen, die sich normalerweise in Strukturen befinden, ohne dass Sie die Strukturen selbst widerspiegeln. Andererseits kann aber auch gesagt werden, dass es sich bei diesen Elementen um echte Ansichtseigenschaften handelt. Sie zeigen grundlegende Merkmale der Ansicht auf und verdienen es daher, als Eigenschaften offengelegt zu werden.

Rezept 3.5 enthält eine komplette Kategorie von Hilfsfunktionen für Ansichtsrahmen. Es ist Ihre Entscheidung, ob Sie diese Eigenschaften verwenden möchten oder nicht. Dabei werden Transformationen nicht berücksichtigt. Diese Eigenschaften sind für einfache Ansichten ohne affine Transformationen gedacht.

```
@interface UIView (ViewFrameGeometry)
@property CGPoint origin;
@property CGSize size;
@property (readonly) CGPoint midpoint;

// topLeft ist synonym mit origin und wird daher hier nicht aufgeführt
@property (readonly) CGPoint bottomLeft;
@property (readonly) CGPoint bottomRight;
@property (readonly) CGPoint topRight;

@property CGFloat height;
@property CGFloat width;
@property CGFloat top;
@property CGFloat left;
@property CGFloat bottom;
@property CGFloat right;

- (void) moveBy: (CGPoint) delta;
- (void) scaleBy: (CGFloat) scaleFactor;
- (void) fitInSize: (CGSize) aSize;
@end
```

```
@implementation UIView (ViewGeometry)
// Ruft den Ursprung ab und legt ihn fest
- (CGPoint) origin
{
   return self.frame.origin;
}

- (void) setOrigin: (CGPoint) aPoint
{
   CGRect newframe = self.frame;
   newframe.origin = aPoint;
   self.frame = newframe;
}

// Ruft die Größe ab und legt sie fest
- (CGSize) size
{
   return self.frame.size;
}

- (void) setSize: (CGSize) aSize
{
   CGRect newframe = self.frame;
   newframe.size = aSize;
   self.frame = newframe;
}

// Fragt andere Rahmenpositionen ab
- (CGPoint) midpoint
{
   // midpoint wird mit Bezug auf das eigene Koordinatensystem der Ansicht
   // bestimmt, center dagegen im Koordinatensystem des Elternelements
   CGFloat x = CGRectGetMidX(self.bounds);
   CGFloat y = CGRectGetMidY(self.bounds);
   return CGPointMake(x, y);
}

- (CGPoint) bottomRight
{
   CGFloat x = self.frame.origin.x + self.frame.size.width;
   CGFloat y = self.frame.origin.y + self.frame.size.height;
   return CGPointMake(x, y);
}
```

```
- (CGPoint) bottomLeft
{
   CGFloat x = self.frame.origin.x;
   CGFloat y = self.frame.origin.y + self.frame.size.height;
   return CGPointMake(x, y);
}

- (CGPoint) topRight
{
   CGFloat x = self.frame.origin.x + self.frame.size.width;
   CGFloat y = self.frame.origin.y;
   return CGPointMake(x, y);
}

// Ruft Höhe, Breite, obere, untere, linke und rechte Position ab
// und legt sie fest
- (CGFloat) height
{
   return self.frame.size.height;
}

- (void) setHeight: (CGFloat) newheight
{
   CGRect newframe = self.frame;
   newframe.size.height = newheight;
   self.frame = newframe;
}

- (CGFloat) width
{
   return self.frame.size.width;
}

- (void) setWidth: (CGFloat) newwidth
{
   CGRect newframe = self.frame;
   newframe.size.width = newwidth;
   self.frame = newframe;
}

- (CGFloat) top
{
   return self.frame.origin.y;
}
```

```
- (void) setTop: (CGFloat) newtop
{
   CGRect newframe = self.frame;
   newframe.origin.y = newtop;
   self.frame = newframe;
}

- (CGFloat) left
{
   return self.frame.origin.x;
}

- (void) setLeft: (CGFloat) newleft
{
   CGRect newframe = self.frame;
   newframe.origin.x = newleft;
   self.frame = newframe;
}

- (CGFloat) bottom
{
   return self.frame.origin.y + self.frame.size.height;
}

- (void) setBottom: (CGFloat) newbottom
{
   CGRect newframe = self.frame;
   newframe.origin.y = newbottom - self.frame.size.height;
   self.frame = newframe;
}

- (CGFloat) right
{
   return self.frame.origin.x + self.frame.size.width;
}

- (void) setRight: (CGFloat) newright
{
   CGFloat delta = newright -
      (self.frame.origin.x + self.frame.size.width);
   CGRect newframe = self.frame;
   newframe.origin.x += delta;
   self.frame = newframe;
}
@end
```

▶ *Rezept 3.5: Kategorie von Hilfsfunktionen für Ansichtsrahmen*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.9 Eine Ansicht mit Grenzen zufällig verschieben

Wenn Sie eine Ansicht an einen zufälligen Punkt verschieben, müssen Sie verschiedene Aspekte bedenken. Sehr oft muss eine Ansicht komplett in den Container ihrer Elternansicht passen, damit nichts abgeschnitten wird. Häufig fügen Sie dem Container auch eine Grenze hinzu, damit die Ansicht die Kante des Elternelements nicht berührt. Wenn Sie mit der vorgefertigten SDK-Version der Klasse `UIView` arbeiten, müssen Sie außerdem zufällige Mittelpunkte verwenden, nicht zufällige Positionen, wie Sie weiter vorn in diesem Kapitel schon gesehen haben. Einfach einen Punkt irgendwo in der Elternansicht auszuwählen, wird diesen Bedingungen meistens nicht gerecht.

Um dieses Problem zu lösen, wird in *Rezept 3.6* eine Reihe von Abständen zwischen den Kanten der Kind- und der Elternansicht definiert. Mit der Struktur `UIEdgeInset` aus dem UIKit des iOS SDK werden die Grenzen der Ansicht definiert. Diese Struktur enthält vier Einsprungwerte, die angeben, wie weit das Rechteck oben, links, unten und rechts gegenüber dem Elterncontainer nach innen versetzt ist.

```
typedef struct {
   CGFloat top, left, bottom, right;
} UIEdgeInsets;
```

Die im Rezept vorgestellte Methode verwendet die Funktion `UIEdgeInsetsInsetRect()`, um ein `CGRect`-Rechteck zu schrumpfen und daraus einen inneren Container zu erstellen, der hier `innerRect` heißt.

Anschließend wird der Container noch weiter verkleinert. Die Grenzen des Rechtecks werden um die Hälfte der Höhe und Breite des Kindelements nach innen gezogen. Dadurch bleibt um jeden Punkt des Unterrechtecks genug Platz für die Kindansicht, ohne dass diese Ansicht mit dem inneren, mit einem Rand versehenen Rechteck überlappt. Jeder beliebige Punkt innerhalb des Unterrechtecks ist ein gültiger Mittelpunkt für die Kindansicht.

```
- (CGPoint) randomCenterInView: (UIView *) aView
   withInsets: (UIEdgeInsets) insets
{
   // Verschiebt die Position um den Einsprungwert und dann um die Größe
   // der Unteransicht nach innen
   CGRect innerRect = UIEdgeInsetsInsetRect([aView bounds], insets);
   CGRect subRect = CGRectInset(innerRect,
      self.frame.size.width / 2.0f, self.frame.size.height / 2.0f);
```

```
    // Gibt einen zufälligen Punkt zurück
    float rx = (float)(random() % (int)floor(subRect.size.width));
    float ry = (float)(random() % (int)floor(subRect.size.height));
    return CGPointMake(rx + subRect.origin.x, ry + subRect.origin.y);
}

- (CGPoint) randomCenterInView: (UIView *) aView
    withInset: (float) inset
{
    UIEdgeInsets insets = UIEdgeInsetsMake(inset, inset, inset, inset);
    return [self randomCenterInView:aView withInsets:insets];
}

- (void) moveToRandomLocationInView: (UIView *) aView

animated: (BOOL) animated

{

if (!animated)

        {
    self.center = [self randomCenterInView:aView withInset:5];

}


     [UIView animateWithDuration:0.3f animations:^(void){

          self.center = [self randomCenterInView:aView withInset:5];}];



}
```

▶ *Rezept 3.6: Eine begrenzte Ansicht zufällig verschieben*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.10 Rezept: Ansichten transformieren

Mit affinen Transformationen können Sie die Geometrie eines Objekts ändern, indem Sie es von einem Koordinatensystem auf ein anderes abbilden. Das iOS SDK unterstützt standardmäßig affine 2D-Transformationen vollständig. Mit ihnen können Sie Ihre Ansichten skalieren, verschieben, drehen und scheren, wie immer Sie es wünschen oder wie Ihre Anwendung es erfordert.

Transformationen sind in Core Graphics definiert und bestehen aus Aufrufen wie `CGAffineTransformMakeRotation` und `CGAffineTransformScale`. Sie erstellen und ändern die 3×3-Transformationsmatrizen. Verwenden Sie den Aufruf `setTransform:` von `UIView`, sobald die Transformation erstellt ist, um zweidimensionale affine Transformationen auf `UIView`-Objekte anzuwenden.

Rezept 3.7 zeigt, wie Sie eine affine Transformation einer `UIView` erstellen und anwenden. Ich habe das Beispiel einfach gehalten. Es gibt einen `NSTimer`, der 30-mal pro Sekunde tickt. Bei jedem Tick wird eine Ansicht um ein Hundertstel von p gedreht und mit einer Kosinuskurve skaliert. Aus zwei Gründen verwende ich den Absolutbetrag des Kosinus: Die Ansicht ist so jederzeit sichtbar und bietet einen hübschen Federeffekt, wenn die Skalierung die Richtung ändert. Das erzeugt eine Drehung und eine ungedämpfte Federanimation.

Dies ist eines der Beispiele, die Sie am besten erstellen und anschauen, während Sie den Code lesen. Sie können dann besser sehen, wie die Methode `handleTimer:` mit den optischen Effekten in Beziehung steht, die Sie sehen.

Hinweis

In diesem Rezept wird die C-Standardbibliothek `math` verwendet, die sowohl die Kosinusfunktion als auch die Konstante `M_PI` enthält.

```
- (void) move: (NSTimer *) aTimer
{
   // Dreht die Ansicht bei jeder Iteration um 1/100 PI
   CGFloat angle = theta * (M_PI / 100.0f);
   CGAffineTransform transform = CGAffineTransformMakeRotation(angle);

   // Theta bewegt sich im Bereich von 0 und 2*PI
   theta = (theta + 1) % 200;

   // Skaliert die Ansicht spaßeshalber mit dem Absolutwert des Kosinus
   float degree = cos(angle);
   if (degree < 0.0) degree *= -1.0f;
   degree += 0.5f;

   // Fügt zur Rotationstransformation eine Skalierung hinzu
   CGAffineTransform scaled =
      CGAffineTransformScale(transform, degree, degree);
```

```
    // Wendet die affine Transformation an
    imageView.transform = scaled;
}
```

▶ *Rezept 3.7: Beispiel für die affine Transformation einer* `UIView`*-Ansicht*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.11 Anzeige- und Interaktionsaspekte

Neben dem physischen Bildschirmlayout stellt die Klasse `UIView` auch Eigenschaften bereit, die festlegen, wie Ansichten auf dem Bildschirm erscheinen und ob die Benutzer mit ihnen interagieren können. Für jede Ansicht gibt es einen Durchsichtigkeitsfaktor (Alpha), der von opak bis transparent reicht. Diesen Faktor können Sie mit `[myView setAlpha:value]` einstellen, wobei die Alphawerte zwischen 0,0 (völlig transparent) und 1,0 (völlig undurchsichtig) liegen. Damit haben Sie eine hervorragende Möglichkeit, um Ansichten zu verbergen und sie auf dem Bildschirm ein- und auszublenden.

Sie können dem Hintergrund jeder Ansicht eine Farbe zuweisen. Beispielsweise färben Sie eine Ansicht mit der folgenden Eigenschaft rot:

```
myView.backgroundColor: [UIColor redColor]
```

Die Eigenschaft wirkt sich auf die verschiedenen Ansichtsklassen jeweils unterschiedlich aus, was davon abhängt, ob die Klassen über Unteransichten verfügen, die den Hintergrund verstellen. Einen transparenten Hintergrund erreichen Sie, indem Sie die Hintergrundfarbe der Ansicht auf farblos einstellen:

```
myView.backgroundColor = [UIColor clearColor];
```

Unabhängig davon, ob Sie den Hintergrund sehen können, hat jede Ansicht eine Eigenschaft für die Hintergrundfarbe. Eine gute Möglichkeit, um die tatsächliche Ausdehnung einer Ansicht zu erkennen, besteht darin, leuchtende, kontrastierende Hintergrundfarben zu verwenden. Wenn Sie erst mit der Entwicklung für iOS beginnen, erlangen Sie durch das Einfärben der Ansichten einen Begriff davon, was auf dem Bildschirm sichtbar ist und was nicht und wo sich die einzelnen Komponenten befinden.

Die Eigenschaft `userInteractionEnabled` legt fest, ob Benutzer eine gegebene Ansicht antippen und mit ihr interagieren können. Bei den meisten Ansichten weist diese Eigenschaft den Standardwert `YES` auf, bei `UIImageView` jedoch `NO`, was neue Entwickler zur Verzweiflung treiben kann. Häufig verwenden sie eine `UIImageView` als Hintergrund und verstehen nicht, warum ihre Schalter, Textfelder und Schaltflächen nicht funktionieren. Aktivieren Sie diese Eigenschaft für alle Ansichten,

die die Benutzer antippen müssen – und deren Unteransichten wie Schaltflächen, Schalter, Auswahlfelder und andere Steuerelemente die Benutzer antippen müssen. Wenn Elemente nicht auf Berührungen reagieren, sollten Sie den Wert der Eigenschaft `userInteractionEnabled` für dieses Element und seine Elternelemente überprüfen.

Deaktivieren Sie diese Eigenschaft für alle Ansichten innerhalb des Interaktionsbereichs, die lediglich der Anzeige dienen. Um beispielsweise auf einer transparenten, bildschirmfüllenden Ansicht eine nicht interaktive Uhr anzuzeigen, schalten Sie deren Interaktion ab. Dadurch werden Berührungen durch diese Ansicht hindurch an den darunterliegenden Interaktionsbereich der Anwendung übergeben. Wollen Sie dagegen eine Blockieransicht im »Bitte-warten-Stil« erstellen, müssen Sie die Benutzerinteraktion für die Überlagerung aktivieren. Dadurch werden alle Berührungen abgefangen und die Benutzer daran gehindert, auf die Hauptschnittstelle unter der Überlagerung zuzugreifen.

Sie können diese Eigenschaft auch bei Übergängen deaktivieren, damit Berührungen keine Aktionen auslösen, während die Ansichten animiert werden. Unerwünschte Bewegungen können vor allem bei Spielen und Rätseln ein Problem darstellen.

3.12 UIView-Animationen

`UIView`-Animationen bieten einen der sonderbaren, aber großartigen Nebeneffekte bei der Arbeit mit iOS als Entwicklungsplattform. Sie ermöglichen es Ihnen, die grafischen Änderungen bei der Aktualisierung von Ansichten fließend darzustellen, was zu weichen, animierten Übergängen führt, die für den Benutzer angenehmer sind. Und das Beste ist, dass all dieses geschieht, ohne dass Sie viel dazu tun müssen.

`UIView`-Animationen sind ideal dazu geeignet, eine optische Brücke zwischen dem aktuellen und dem geänderten Zustand einer Ansicht zu bauen. Damit heben Sie optische Änderungen hervor und verbinden diese Änderungen. Die folgenden Änderungen sind animierbar:

- **Änderungen der Position** Verschieben einer Ansicht auf dem Bildschirm
- **Änderungen der Größe** Aktualisierungen des Ansichtsrahmens
- **Änderungen der Dehnung** Aktualisierungen der Dehnbereiche im Inhalt einer Ansicht
- **Änderungen der Transparenz** Änderung des Alphawerts der Ansicht
- **Änderungen des Status** Verborgen oder sichtbar
- **Änderungen der Ansichtsreihenfolge** Änderungen daran, welche Ansicht vorn liegt
- **Änderungen der Drehung** Das gilt auch für alle anderen affinen Transformationen, die Sie auf eine Ansicht anwenden.

Zwischen den SDK-Versionen 3.x und 4.x wurden die Animationen einer erheblichen Überarbeitung unterzogen. Mit Version 4.x haben die Entwickler eine Möglichkeit bekommen, die in Objective-C neu eingeführten Blöcke zu nutzen, um sich Animationsaufgaben zu erleichtern. Sie können zwar immer noch mit den ursprünglichen Techniken für animierte Transformationen arbeiten, allerdings bieten die neuen Alternativen eine viel einfachere Vorgehensweise. In den nächsten Abschnitten sehen wir uns den ursprünglichen Ansatz und die Neuerungen an.

3.12.1 UIView-Animationstransaktionen erstellen

In ihrer alten Form ohne Blöcke funktionierten `UIView`-Animationen als Transaktionen, also als eine Folge von Operationen, die auf einmal ausgeführt werden. Eine solche Transaktion starten Sie mit `beginAnimations:context:` und beenden sie mit `commitAnimations`. Für die Transaktion können Sie auch Animationsoptionen festlegen. Diese Klassenmethoden senden Sie an `UIView` und nicht an einzelne Ansichten. Zwischen dem `begin`- und dem `commit`-Aufruf definieren Sie, wie die Animation abläuft, und führen die tatsächlichen Aktualisierungen der Ansicht durch. Sie verwenden dabei die folgenden Animationssteuerungen:

- `beginAnimations:context:` Kennzeichnet den Start der Animationstransaktion.
- `setAnimationCurve` Definiert die Art und Weise, wie die Animation beschleunigt und abgebremst wird. Verwenden Sie `ease-in/ease-out` (`UIViewAnimationCurveEaseInOut`), sofern Sie keinen zwingenden Grund haben, eine andere Kurve auszuwählen. Die anderen Kurventypen sind `ease in` (in die Animation beschleunigen), `linear` (keine Animationsbeschleunigung) und `ease out` (aus der Animation beschleunigen). `Ease-in/ease-out` bietet den natürlichsten Animationsstil.
- `setAnimationDuration` Gibt die Länge der Animation in Sekunden an. Das ist ein wirklich nützlicher Parameter. Sie können die Animation so lange dehnen, wie es erforderlich ist. Stellen Sie jedoch nicht die Geduld des Benutzers auf die Probe, sondern halten Sie die Dauer Ihrer Animationen unter einer oder zwei Sekunden. Zum Vergleich: Wenn die Tastatur auf den Bildschirm geschoben oder von ihm entfernt wird, dauert die Animation 0,3 Sekunden.
- `commitAnimations` Kennzeichnet das Ende der Animationstransaktion.

Fügen Sie die eigentlichen Befehle für die Ansichtsänderungen nach dem Festlegen der Einzelheiten und vor dem Ende der Animation ein:

```
CGContextRef context = UIGraphicsGetCurrentContext();
[UIView beginAnimations:nil context:context];
[UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
[UIView setAnimationDuration:1.0f];

// Hier stehen die Ansichtsänderungen
[contentView setAlpha:0.0f];

[UIView commitAnimations];
```

Dieses Listing zeigt `UIView`-Animationen im Einsatz. Hier werden eine Animationskurve und die Animationsdauer gesetzt (in diesem Fall eine Sekunde). Die eigentliche animierte Änderung ist eine Aktualisierung der Transparenz. Der Alphawert der Inhaltsansicht geht gegen null und macht sie unsichtbar. Die Ansicht verschwindet nicht einfach, sondern die Animationstransaktion verlangsamt die Änderung und blendet sie aus. Beachten Sie den Aufruf von `UIGraphicsGetCurrentContext()`, der den grafischen Kontext an der Spitze des aktuellen Ansichtsstacks zurückgibt. Ein

Grafikkontext bietet eine virtuelle Verbindung zwischen den abstrakten Aufrufen zum Zeichnen und den Pixeln auf dem Bildschirm (bzw. innerhalb eines Bildes). Für dieses Argument können Sie im jüngsten SDK auch `nil` übergeben, ohne dass dies Schwierigkeiten nach sich zieht.

Transaktionsgestützte Animationen von Ansichten können einen optionalen Delegate über Statusänderungen informieren, also darüber, dass die Animation begonnen hat oder abgeschlossen ist. (Wie Sie gleich sehen werden, lässt sich das mithilfe von Blöcken viel einfacher bewerkstelligen.) Dadurch können Sie das Ende einer Animation abfangen, um die nächste Animation in einer Folge zu starten oder um andere Aufgaben durchzuführen, die nach der Animation anstehen. Den Delegate legen Sie wie folgt mit `setAnimationDelegate:` fest:

```
[UIView setAnimationDelegate:self];
```

Um einen Callback am Ende der Animation vorzusehen, geben Sie den Selektor an, der an den Delegate gesendet wird:

```
[UIView setAnimationDidStopSelector:finished:context:)];
```

Hinweis

Die meisten Apple-eigenen Animationen dauern etwa eine drittel oder eine halbe Sekunde. Wenn Sie mit Hilfsansichten auf dem Bildschirm arbeiten (die ähnliche unterstützende Aufgaben erfüllen wie die Tastatur und die Warneinblendungen von Apple), sollten Sie die Dauer der Animationen an diese Elemente anpassen.

3.12.2 Animationen mit Blöcken erstellen

Blockkonstrukte erleichtern es, grundlegende Animationseffekte im Code zu erstellen. Das folgende Beispiel sorgt für den gleichen Ausblendevorgang wie der vorherige Code, aber mit einer Struktur, die der Intuition viel stärker entgegenkommt:

```
[UIView animationWithDuration: 1.0f
   animations:^{contentView.alpha = 0.0f;}];
```

Statt mit sechs wird der Effekt hier nur mit einer Anweisung und einem eingebetteten Block erzielt. Die Animationskurve folgt dem Standardverlauf »ease-in/ease-out«.

Auch die Ergänzung um einen Vervollständigungsblock wird viel einfacher. Mit dem folgenden Code wird die Inhaltsansicht ausgeblendet und nach Abschluss der Animation aus der übergeordneten Ansicht entfernt:

```
[UIView animationWithDuration: 1.0f
   animations:^{contentView.alpha = 0.0f;}
   completion:^(BOOL done){[contentView removeFromSuperview];}];
```

Nach dem Übergang von der alten Vorgehensweise zu derjenigen mit Blöcken werden Sie die schlichte Eleganz der neuen Implementierung erkennen. Wenn Sie weitere Optionen für Ihre Animationen benötigen, können Sie mit der blockgestützten Rundum-Methode `animateWithDuration:delay:options:animations:completion:` sowohl Animationsoptionen (als Maske) übergeben als auch die Animation verzögern (was eine einfache Möglichkeit zur Verkettung bildet).

3.12.3 Bedingte Animation

In den seltenen Fällen, in denen ich mit bedingten Animationen arbeite – wenn die Animation selbst also optional ist –, verwende ich weiterhin herkömmliche Transaktionen anstelle von Blöcken. In dem folgenden Beispiel wären Blöcke weniger effizient als diese `UIView`-Animation alten Stils:

```
if (animated)
{
   [UIView beginAnimations:@"SwitchAnimation" context:nil];
   [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
   [UIView setAnimationDuration:0.1f];
}

if (self.on)
   rect.origin = CGPointMake(0.0f, 0.0f);
else
   rect.origin = CGPointMake(newX, 0.0f);
backdrop.frame = rect;

if (animated) [UIView commitAnimations];
```

Es wäre hier zwar auch möglich, eine Überprüfung auf Animationen durchzuführen und dann entweder einen Animationsblock aufzurufen oder den Code separat auszuführen, doch wie Sie hier sehen, ist es in diesem Fall sehr viel einfacher, `if`-Anweisungen und herkömmliche Animationsaufrufe zu verwenden.

3.13 Rezept: Ansichten ein- und ausblenden

Es kann vorkommen, dass Sie Informationen auf dem Bildschirm anzeigen möchten, die eine Ansicht überlagern, aber selbst nichts tun. Sie können z. B. eine Bestenliste, irgendwelche Anweisungen oder eine kontextsensitive Hilfe ausgeben. *Rezept 3.8* zeigt, wie Sie einen `UIView`-Animationsblock verwenden, um eine Ansicht ein- und auszublenden. Dieses Rezept folgt dem ganz grundlegenden Animationsansatz. Es erstellt einen umgebenden Animationsblock, der die Eigenschaft `alpha` festlegt.

Beachten Sie, wie dieser Code das Verhalten der rechten Leistenschaltfläche steuert. Beim Antippen wird sie sofort deaktiviert, bis die Animation abgeschlossen ist. Der Vervollständigungsblock der Animation reaktiviert die Schaltfläche wieder und schaltet den Schaltflächentext und den Callbackselektor in den gegenteiligen Zustand um. Dadurch kann die Animation mit dieser Schaltfläche im eingeschalteten Zustand ausgeschaltet und im ausgeschalteten Zustand eingeschaltet werden.

```
- (void) fadeOut: (id) sender
{
   self.navigationItem.rightBarButtonItem.enabled = NO;
   [UIView animateWithDuration:1.0f
      animations:^{
         // Hier findet der eigentliche Ausblendevorgang statt
         imageView.alpha = 0.0f;
      }
      completion:^(BOOL done){
         self.navigationItem.rightBarButtonItem.enabled = YES;
         self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Fade In",@selector(fadeIn:));
      }];
}
- (void) fadeIn: (id) sender
{
   self.navigationItem.rightBarButtonItem.enabled = NO;
   [UIView animateWithDuration:1.0f
      animations:^{
         // Hier findet der Einblendevorgang statt
         imageView.alpha = 1.0f;
      }
      completion:^(BOOL done){
         self.navigationItem.rightBarButtonItem.enabled = YES;
         self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Fade Out",@selector(fadeOut:));
      }];
}

- (void) loadView

{

    [super loadView];

   self.navigationItem.rightBarButtonItem =
      BARBUTTON(@"Fade Out",@selector(fadeOut:));

   imageView = [[UIImageView alloc] initWithImage:
      [UIImage imageNamed:@"BFlyCircle.png"]];
   [self.view addSubview:imageView];
}
```

▶ *Rezept 3.8: Änderungen der Transparenz mithilfe der Eigenschaft* `alpha` *einer Ansicht animieren*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.14 Rezept: Ansichten austauschen

Der `UIView`-Animationsblock beschränkt Sie nicht auf eine Änderung. *Rezept 3.9* verbindet Größenänderungen mit Änderungen der Transparenz, um eine überzeugendere Animation zu erstellen. Sie erreichen das, indem Sie mehrere Anweisungen gleichzeitig zum Animationsblock hinzufügen. Der Rezeptcode führt fünf Aktionen gleichzeitig durch. Er zoomt und blendet eine Ansicht ein, während er eine andere auszoomt und ausblendet. Anschließend tauscht er die beiden in der Arrayliste der Unteransichten aus.

Bereiten Sie das Objekt `back` zur Animation vor, indem Sie es verkleinern und transparent machen. Bei der ersten Ausführung der Methode `swap:` ist diese Ansicht bereit zum Einblenden und zur Änderung der Größe. Wie in *Rezept 3.8* reaktiviert der Vervollständigungsblock die rechte Schaltfläche in der Leiste, sodass es möglich ist, mehrmals hintereinander darauf zu tippen.

```
@implementation TestBedViewController
- (void) swap: (id) sender
{
   self.navigationItem.rightBarButtonItem.enabled = NO;
   [UIView animateWithDuration:1.0f
      animations:^{
         frontObject.alpha = 0.0f;
         backObject.alpha = 1.0f;
         frontObject.transform = CGAffineTransformMakeScale(0.25f,
0.25f);
         backObject.transform = CGAffineTransformIdentity;
         [self.view exchangeSubviewAtIndex:0
            withSubviewAtIndex:1];
      }
      completion:^(BOOL done){
         self.navigationItem.rightBarButtonItem.enabled = YES;

         // Ansichtsverweise austauschen
         UIImageView *tmp = frontObject;
         frontObject = backObject;
         backObject = tmp;
   }];
}
```

► *Rezept 3.9: Mehrere Ansichtenänderungen in einem Animationsblock kombinieren*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.15 Rezept: Ansichten wenden

Mit Übergängen (*Transitions*) können Sie Ihre `UIView`-Animationsblöcke erweitern, um noch mehr optisches Flair zu bieten. Die vier Übergangsarten `UIViewAnimationTransitionFlipFromLeft`, `UIViewAnimationTransitionFlipFromRight`, `UIViewAnimationTransitionCurlUp` und `UIViewAnimationTransitionCurlDown` tun genau das, was ihre Namen andeuten. Damit können Sie Ansichten nach links oder rechts wenden oder wie in der Anwendung *Karten* nach oben oder unten aufblättern. Wie das geht, zeigt *Rezept 3.10*.

Rezept 3.10 nutzt die blockgestützte API für `transitionFormView:toView:duration:options:completion:`. Diese Methode ersetzt eine Ansicht, indem sie sie aus ihrer übergeordneten Ansicht entfernt und die neue Ansicht zum Elternobjekt der ursprünglichen hinzufügt. Sie animiert diesen Vorgang während des angegebenen Zeitraums mit dem unter `options:` angegebenen Übergang. In *Rezept 3.10* wird die Ansicht von links nach rechts gewendet, doch können Sie hier bei Bedarf auch einen der anderen drei Übergänge einsetzen.

Übergeordnete Ansichten behalten normalerweise die Ansichten bei, die ihnen hinzugefügt werden, und geben sie beim Entfernen wieder frei. Hier müssen Sie sorgfältig darauf achten, dass Sie die entfernte Ansicht beibehalten, wenn Sie sie später wieder hinzufügen möchten.

```
- (void) flip: (id) sender
{
   self.navigationItem.rightBarButtonItem.enabled = NO;
   [UIView transitionFromView: fromPurple ? purple : maroon
      toView: fromPurple ? maroon : purple
      duration: 1.0f
      options: seg.selectedSegmentIndex ? UIViewAnimationOptionTransiti-
onCurlUp :UIViewAnimationOptionTransitionFlipFromLeft
      completion: ^(BOOL done){
         self.navigationItem.rightBarButtonItem.enabled = YES;
         fromPurple = !fromPurple;
      }];
}
```

▶ *Rezept 3.10: Übergänge mit* `UIView`*-Animationen*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.16 Rezept: Core Animation-Übergänge verwenden

Neben `UIView`-Animationen unterstützt iOS im Rahmen des Quartz Core-Frameworks auch *Core Animation*. Die Core Animation-API bietet hochgradig flexible Animationslösungen für iOS-Anwendungen. Vor allem enthält sie Übergänge für die gleichen Wechsel von einer Ansicht zur anderen, die Sie im vorherigen Rezept kennengelernt haben.

Core Animation-Übergänge erweitern Ihre Möglichkeiten für `UIView`-Animationen mit nur geringen Unterschieden bei der Implementierung. `CATransition`s wirken sich auf Ebenen statt auf Ansichten aus. Ebenen sind die Rendering-Oberflächen von Core Animation, die mit den einzelnen `UIView`s verknüpft sind. Wenn Sie mit Core Animation arbeiten, wenden Sie `CATransition`s auf die Standardebene einer Ansicht (`[myView layer]`) statt auf die Ansicht selbst an.

Bei diesen Übergängen setzen Sie Ihre Parameter nicht wie bei `UIView`-Animationen in der `UIView`. Stattdessen erstellen Sie ein Core Animation-Objekt, richten dessen Parameter ein und fügen dann den parametrisierten Übergang zur Ebene hinzu.

```
CATransition *animation = [CATransition animation];
animation.delegate = self;
animation.duration = 1.0f;
animation.type = kCATransitionMoveIn;
animation.subtype = kCATransitionFromTop;

// Hier werden Ansichten ausgetauscht oder entfernt

[myView.layer addAnimation:animation forKey:@"move in"];
```

Animationen haben sowohl einen *Typ* als auch einen *Untertyp*. Ersterer gibt die Art des verwendeten Übergangs an, Letzterer dessen Richtung. Zusammen legen der Typ und der Untertyp fest, wie sich die Ansichten verhalten sollen, wenn die Animation auf sie angewendet wird.

Core Animation-Übergänge unterscheiden sich von den oben behandelten `UIViewAnimationTransition`-Rezepten. Cocoa Touch bietet vier Arten solcher Animationen, die in *Rezept 3.11* vorgestellt werden. Zu den verfügbaren Typen gehören Überblendungen, das Wegschieben (eine Ansicht schiebt die andere vom Bildschirm), Aufdeckungen (eine Ansicht gleitet von einer anderen herunter) und Bedeckungen (eine Ansicht gleitet über eine andere). Bei den letzten drei Typen können Sie die Bewegungsrichtung für den Übergang durch Untertypen angeben. Es ist offensichtlich, dass Überblendungen keine Richtung haben und daher auch keine Untertypen aufweisen.

3.16 Rezept: Core Animation-Übergänge verwenden

Da Core Animation ein Bestandteil des Quartz Core-Frameworks ist, müssen Sie dieses Framework zu Ihrem Projekt hinzufügen und `<QuartzCore/QuartzCore.h>` in den Code importieren, wenn Sie diese Merkmale verwenden möchten.

Hinweis

Core Animation von Apple enthält 2D- und 3D-Routinen, die auf Objective-C-Klassen beruhen. Diese Klassen bieten Grafikrendering und Animationen für iOS- und Macintosh-Anwendungen. Core Animation umgeht viele Low-Level-Entwicklungsaufgaben, die z. B. mit OpenGL verbunden sind, sodass die Arbeit damit so einfach wie mit hierarchischen Ansichten ist.

```
- (void) animate: (id) sender
{
   // Die Animation einrichten
   CATransition *animation = [CATransition animation];
   animation.delegate = self;
   animation.duration = 1.0f;

   switch ([(UISegmentedControl *)self.navigationItem.titleView
               selectedSegmentIndex])
   {
      case 0:
         animation.type = kCATransitionFade;
         break;
      case 1:
         animation.type = kCATransitionMoveIn;
         break;
      case 2:
         animation.type = kCATransitionPush;
         break;
      case 3:
         animation.type = kCATransitionReveal;
      default:
         break;
   }
   animation.subtype = kCATransitionFromLeft;

   // Die Animation durchführen
   [self.view exchangeSubviewAtIndex:0 withSubviewAtIndex:1];
   [self.view.layer addAnimation:animation forKey:@"animation"];
}
```

▶ *Rezept 3.11: Übergänge mit Core Animation animieren*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.17 Rezept: Ansichten beim Erscheinen nachfedern lassen

Häufig verwendet Apple zwei Animationsblöcke, bei denen der zweite nach dem Abschluss des ersten aufgerufen wird, um eine Animation nachfedern zu lassen. Beispielsweise wird eine Ansicht etwas stärker vergrößert als notwendig und dann mit der zweiten Animation auf die endgültige Größe gebracht. Durch dieses »Nachfedern« werden Animationen lebendiger und wirken realer.

Wenn Sie eine Animation nach einer anderen aufrufen, müssen Sie sicherstellen, dass sich die beiden Animationen nicht überschneiden. In der früheren Ausgabe dieses Buches wurden blockierende modale Animationen behandelt, doch seit in iOS die Unterstützung für Vervollständigungsblöcke eingeführt wurde, ist es jetzt viel einfacher, einen Satz verschachtelter Animationsblöcke mit verketteten Animationen im Vervollständigungsblock zu verwenden. In *Rezept 3.12* wird diese neue Vorgehensweise eingesetzt, um Ansichten ein klein wenig größer zu machen als im Endzustand und sie dann in den vorgesehenen Rahmen hineinschrumpfen zu lassen.

In diesem Rezept werden zwei einfache `typedef`s verwendet, um die Deklaration der einzelnen Animationen und Vervollständigungsblöcke zu vereinfachen. Beachten Sie, dass die Animationsblockphasen, die die Skalierung der jeweiligen Ansicht übernehmen, in der richtigen Reihenfolge definiert sind. Der erste Block verkleinert die Ansicht, der zweite macht sie größer, und der dritte stellt die ursprüngliche Größe wieder her.

Die Vervollständigungsblöcke gehen genau anders herum vor. Da jeder Block von dem vorausgehenden abhängt, müssen Sie sie in umgekehrter Reihenfolge erstellen. Beginnen Sie mit den endgültigen Nebenwirkungen, und arbeiten Sie sich bis zum Original zurück. In *Rezept 3.12* hängt `bounceLarge` von `shrinkBack` ab, das wiederum von `reenable` abhängig ist. Diese umgekehrte Definitionsreihenfolge kann ein bisschen knifflig sein, ist auf jeden Fall aber besser, als den gesamten Code in verschachtelten Blöcken zu gestalten.

```
typedef void (^AnimationBlock)(void);
typedef void (^CompletionBlock)(BOOL finished);

- (void) bounce: (id) sender
{
   self.navigationItem.rightBarButtonItem.enabled = NO;

   // Definiert die drei Animationsphasen in fortlaufender Reihenfolge
   AnimationBlock makeSmall = ^(void){
      bounceView.transform = CGAffineTransformMakeScale(0.01f, 0.01f);};
```

```
    AnimationBlock makeLarge = ^(void){
        bounceView.transform = CGAffineTransformMakeScale(1.15f, 1.15f);};
    AnimationBlock restoreToOriginal = ^(void) {
        bounceView.transform = CGAffineTransformIdentity;};

    // Erstellt die drei Vervollständigungsblöcke in umgekehrter Reihenfolge
    CompletionBlock reenable = ^(BOOL finished) {
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Start", @selector(bounce:));};
    CompletionBlock shrinkBack = ^(BOOL finished) {
        [UIView animateWithDuration:0.2f
            animations:restoreToOriginal completion: reenable];};
    CompletionBlock bounceLarge = ^(BOOL finished){
        [UIView animateWithDuration:0.2
            animations:makeLarge completion:shrinkBack];};

    // Startet die Animation
    [UIView animateWithDuration: 0.5f
        animations:makeSmall completion:bounceLarge];
}
```

▶ *Rezept 3.12: Nachfedernde Ansichten*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.18 Rezept: Bildansichten animieren

Die Klasse `UIImageView` ermöglicht Ihnen nicht nur, statische Bilder anzuzeigen, sondern enthält auch Animationen dafür. Nachdem Sie ein Array aus Bildern geladen haben, können Sie die `UIImageView`-Instanzen anweisen, die Bilder zu animieren. *Rezept 3.13* zeigt, wie das geht.

Sie beginnen damit, ein Array zu erstellen und mit einzelnen Bildern zu füllen, die Sie aus einer Datei laden. Dann weisen Sie dieses Array der Eigenschaft `animationImages` einer `UIImageView`-Instanz zu. Setzen Sie `animationDuration` auf die Gesamtdauer der Schleife, um alle Bilder in dem Array anzuzeigen. Anschließend starten Sie die Animation, indem Sie die Nachricht `startAnimating` senden. (Es gibt auch die ergänzende Methode `stopAnimating`.)

Wenn Sie die animierte Bildansicht zu Ihrer Schnittstelle hinzufügen, können Sie sie an einer festen Stelle platzieren, aber auch so animieren wie jede andere `UIView`-Instanz.

```
- (void) loadView
{
    [super loadView];

    // Lädt die Schmetterlingsbilder
    NSMutableArray *bflies = [NSMutableArray array];
    for (int i = 1; i <= 17; i++)
        [bflies addObject:[UIImage imageWithContentsOfFile:
            [[NSBundle mainBundle] pathForResource:
                [NSString stringWithFormat:@"bf_%d", i]
ofType:@"png"]]];

    // Erstellt die Ansicht
    butterflyView = [[UIImageView alloc] initWithFrame:
                            CGRectMake(0.0f, 0.0f, 60.0f, 60.0f)];

    // Legt die Animationszellen und die Dauer fest
    butterflyView.animationImages = bflies;
    butterflyView.animationDuration = 0.75f;

     // Fügt die Ansicht zur Elternansicht hinzu und startet die Animation
    [self.view addSubview:butterflyView];
    [butterflyView startAnimating];

    [NSTimer scheduledTimerWithTimeInterval:3.0f target:self
        selector:@selector(updateButterfly:)
        userInfo:nil repeats:YES];
}
```

▶ *Rezept 3.13:* `UIImageView`-*Animationen verwenden*

Den Rezeptcode finden

Den in diesem Rezept verwendeten Code erhalten Sie unter *http://github.com/erica/iOS-5-Cookbook*. Falls Sie das Festplattenimage mit dem gesamten Beispielcode zum Buch heruntergeladen haben, wechseln Sie zum Ordner für Kapitel 6 und öffnen das Projekt zu diesem Rezept.

3.19 Zum guten Schluss: Spiegelungen zu Ansichten hinzufügen

Spiegelungen erhöhen den realistischen Eindruck von Objekten auf dem Bildschirm. Sie geben den üblichen Ansichten über einem Hintergrund zusätzliche optische Würze. Dank der Klasse `CAReplicatorLayer` lassen sich Spiegelungen sehr einfach hervorrufen. Diese Klasse repliziert die Ebene einer Ansicht und ermöglicht es, Transformationen darauf anzuwenden.

Aufgrund der Replikation sind diese Spiegelungen »live«: Alle Änderungen an der Hauptansicht zeigen sich sofort in der Reflexionsebene, was Sie selbst anhand des Beispielcodes zu diesem Kapitel überprüfen können. Sie können scrollbare Textansichten, Webansichten, Schalter usw. hinzufügen. Alle Änderungen an der Originalansicht werden repliziert, sodass ein vollständig passives System für Spiegelungen entsteht.

Gute Spiegelungen werden schwächer, je weiter sie von der ursprünglichen Ansicht entfernt sind. Der folgende Code fügt (mit der Eigenschaft `usesGradientOverlay`) eine optionale Verlaufsüberlagerung hinzu, die die Ansicht an ihrem unteren Ende verschwinden lässt. Als Verlauf wird hier ein `CAGradientLayer` verwendet.

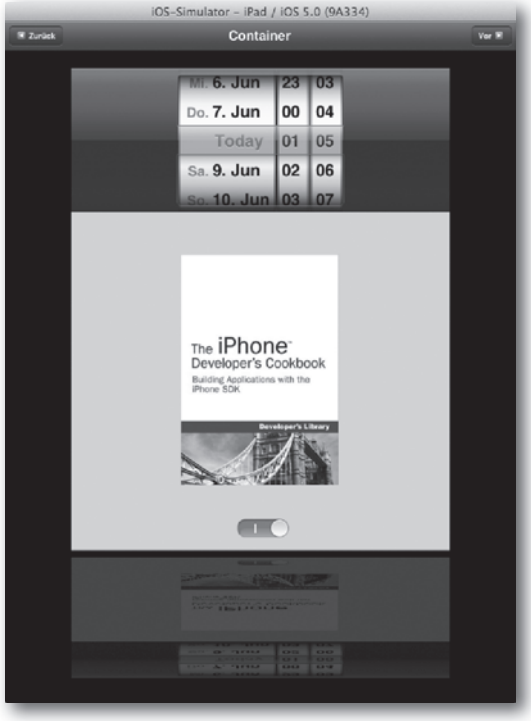
Zwischen dem unteren Rand der Originalansicht und ihrer Spiegelung wird hier ein willkürlicher Abstand gelassen, wie Sie auch in *Abbildung 3.3* sehen. Der Wert beträgt in diesem Beispiel 10 Punkt und ist hartkodiert, Sie können ihn aber nach Belieben anpassen.

```
@implementation ReflectingView
@synthesize usesGradientOverlay;

// Verwenden Sie immer einen Replikator als Grundebene
+ (Class) layerClass
{
   return [CAReplicatorLayer class];
}

// Entfernt alle vorhandenen Verläufe von der Elternansicht
- (void) dealloc
{
   [gradient removeFromSuperlayer];
}

- (void) setupGradient
{
   // Fügt der Elternansicht eine neue Verlaufsebene hinzu
   UIView *parent = self.superview;
   if (!gradient)
```

```
    {
        gradient = [CAGradientLayer layer];
        CGColorRef c1 = [[UIColor blackColor]
            colorWithAlphaComponent:0.5f].CGColor;
        CGColorRef c2 = [[UIColor blackColor]
            colorWithAlphaComponent:0.9f].CGColor;
        [gradient setColors:[NSArray arrayWithObjects:
            (__bridge id)c1, (__bridge id)c2, nil]];
        [parent.layer addSublayer:gradient];
    }

    // Platziert den Verlauf mit der Geometrie der Spiegelung genau
    // unter der Ansicht
    float desiredGap = 10.0f; // Abstand zwischen Ansicht und Reflexion
    CGFloat shrinkFactor = 0.25f; // Reflexionsgröße
    CGFloat height = self.bounds.size.height;
    CGFloat width = self.bounds.size.width;
    CGFloat y = self.frame.origin.y;

    [gradient setAnchorPoint:CGPointMake(0.0f,0.0f)];
    [gradient setFrame:CGRectMake(0.0f, y + height + desiredGap,
        width, height * shrinkFactor)];
    [gradient removeAllAnimations];

}

- (void) setupReflection
{
    CGFloat height = self.bounds.size.height;
    CGFloat shrinkFactor = 0.25f;

    CATransform3D t = CATransform3DMakeScale(1.0, -shrinkFactor, 1.0);

    // Durch die Skalierung wird der Schatten in der Ansicht zentriert.
    // Die Verschiebung erfolgt im Bezugssystem des geschrumpften Objekts.
    float offsetFromBottom = height * ((1.0f - shrinkFactor) / 2.0f);
    float inverse = 1.0 / shrinkFactor;
    float desiredGap = 10.0f;
    t = CATransform3DTranslate(t, 0.0, -offsetFromBottom * inverse
        - height - inverse * desiredGap, 0.0f);
```

```
    CAReplicatorLayer *replicatorLayer = (CAReplicatorLayer*)self.layer;
    replicatorLayer.instanceTransform = t;
    replicatorLayer.instanceCount = 2;

    // Die Verwendung des Verlaufs muss ausdrücklich festgelegt werden
    if (usesGradientOverlay)
       [self setupGradient];
    else
    {
       // Dunkelt die Spiegelung ab, wenn kein Verlauf verwendet wird
       replicatorLayer.instanceRedOffset = -0.75;
       replicatorLayer.instanceGreenOffset = -0.75;
       replicatorLayer.instanceBlueOffset = -0.75;
    }
}
@end
```

▶ *Listing 3.3: Spiegelungen erstellen*



▶ *Abbildung 3.3: Die Spiegelung ist eine replizierte Ebene und passt sich in Echtzeit an Änderungen der Originalansicht an.*

3.20 Zusammenfassung

`UIViews` stellen die Bildschirmkomponenten zur Verfügung, die die Benutzer sehen und mit denen sie interagieren. Wie dieses Kapitel gezeigt hat, bieten sie selbst in ihrer grundlegendsten Form eine unglaubliche Flexibilität und Leistung. Sie haben erfahren, wie Sie Ansichten verwenden, um Elemente auf einem Bildschirm anzuzeigen, wie Sie Ansichten nach Tag oder Name abrufen und wie Sie bemerkenswerte Animationen verwenden. Die folgende Liste führt noch einmal einzelne Punkte der Rezepte auf, die Sie in diesem Kapitel gesehen haben und über die Sie nachdenken sollten, bevor Sie weiterlesen:

- Wenn Sie mehrere Ansichten auf dem Bildschirm haben, sollten Sie immer an die Hierarchie denken. Nutzen Sie Ihr Hierarchievokabular (`bringSubviewToFront:`, `sendSubviewToBack:`, `exchangeSubviewAtIndex:withSubviewAtIndex:`) für Ihre Ansichten, und bieten Sie Ihren Benutzern stets den richtigen optischen Kontext.
- Lassen Sie sich nicht durch den Konflikt zwischen `frame` in Core Graphics und `center` in `UIKit` behindern, sondern verwenden Sie Funktionen, um zwischen diesen Strukturen zu wechseln und die gewünschten Ergebnisse zu erzielen. Das gilt vor allem für einfache Ansichten ohne Transformationen.
- Freunden Sie sich mit Tags an. Sie bieten auf vergleichbare Weise unmittelbaren Zugriff auf Ansichten wie die Symboltabelle eines Programms auf die Variablen. Sie sind nicht Böses oder Gefährliches, sondern eine sinnvolle Ergänzung ihres Entwicklungsvokabulars.
- Blöcke sind etwas Wunderbares. Erleichtern Sie sich damit die Arbeit, vereinfachen Sie Ihren Code und Animationen.
- Animieren Sie alles. Animationen müssen nicht laut, aufdringlich oder geschmacklos sein. Dank der umfangreichen Unterstützung für Animationen im iOS SDK können Sie Benutzeraufgaben sanft ineinander übergehen lassen. Solche unaufdringlichen, fließenden Übergänge machen das iOS-Flair aus. Kurze, sanfte, klare Wechsel sind das A und O in iOS.

Copyright